ADVAN LANGUAGE DESIGNS

BASIC COMPILER

Tutorial
and
Reference Manual

ADVAN LANGUAGE DESIGNS

B A S I C C O M P I L E R

Tutorial
and
Reference Manual

WARNING

DISCLAIMER OF WARRANTY

THIS SOFTWARE AND MANUAL ARE SOLD "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY. THE SELLER'S SALESPERSONS MAY HAVE MADE STATEMENTS ABOUT THIS SOFTWARE. ANY SUCH STATEMENTS DO NOT CONSTITUTE WARRANTIES AND SHALL NOT BE RELIED ON BY THE BUYER IN DECIDING WHETHER TO PURCHASE THIS PROGRAM.

THIS PROGRAM IS SOLD WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES WHATSOEVER. BECAUSE OF THE DIVERSITY OF CONDITIONS AND HARDWARE UNDER WHICH THIS PROGRAM MAY BE USED, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. THE USER IS ADVISED TO TEST THE PROGRAM THOROUGHLY BEFORE RELYING ON IT. THE USER MUST ASSUME THE ENTIRE RISK OF USING THE PROGRAM. ANY LIABILITY OF SELLER OR MANUFACTURER WILL BE LIMITED EXCLUSIVELY TO PRODUCT REPLACEMENT OR REFUND OF THE PURCHASE PRICE.

CONTENTS

REFERENCE MANUAL

Advan BASIC has many features not available in ATARI BASIC and because Advan is a compiler, your programs will normally run faster. To get the most out of Advan BASIC you need to read the tutorial section of this manual. There is also a reference section that describes each command and built-in function.

Main Features

1. Although Advan BASIC is a compiler, it has many of the user friendly features of an interpreter.
   a) You enter programs using the BASIC, not an editor.

   b) Program lines are checked for syntax errors as you enter them; any errors detected are immediately displayed.

   c) In most cases, error messages are given, rather than error numbers.

   d) In most cases, Advan shows the line and position in the line where the error occurred.

   e) Several disk commands are built-in. You can get a directory, or kill, lock, unlock, and rename a file without leaving the BASIC or effecting the program in memory.

2. Advan BASIC supports integer, real, and string variables.
   a) Variable names may be any length.

   b) Strings don't need to be dimensioned.

   c) String arrays, as well as integer and real arrays, are available (up to 64 dimensions).

   d) Real and integer numbers may be mixed in expressions.

   e) The ability to use integers is especially important since they take up only 1/3 as much space, and calculations are at least 3 times faster than those with real variables and real numbers.

3. REPEAT-UNTIL, WHILE-WEND, CASE, IF-THEN-ELSE, and multi-line IF commands are available. Also, program lines can be indented. These commands make it easier to program, and allow many of the techniques used in structured programming.

4. There are special commands for player-missiles.
   a) You can define a figure and insert the figure into a player or missile.

   b) You can set up the program to automatically move a player horizontally or vertically at specified rates. Once started, their positions will be changed automatically during the vertical blank interrupt, so that the program can do other things during their movement.

   c) You can set up the system to automatically change a displayed figure as well as automatically move it. This allows player-missile animation

without the need for machine language code.

5. Advan BASIC uses three commands to take advantage of the ATARI sound capability. One of them lets you set up a tune or a sequence of sounds which the system plays automatically (the program can do other things while the sound is being generated). You can even specify whether the sound will be repeated continuously, played once, or played a given number of times.

6. There are two commands which use the ATARI display list interrupt capability. One lets you insert or remove a display list interrupt. The other lets you change what is done at the interrupt.

7. Functions and named subroutines are available. From zero to four arguments may be used.

8. An assembler is built into Advan BASIC.
   a) Mnemonics can be used for 6502 commands.

   b) You can use line numbers to specify the destinations of jumps and branches.

   c) You can directly access and use BASIC variables in the assembly language code.

   d) The assembly language code can be inserted into a BASIC program. You do not have to worry about where to place the machine language code.

9. There are several utility programs on the Master disks:
   a) Programs to copy files or disks and to format disks.

   b) A program which allows 64K of the 130XE computer's memory to be used as a RAM disk.

   c) A program to check if a variable has been used only once. This is useful in catching misspelled variables.

   d) A checksum program which is useful if you are sending program listings to friends, newsletters, or magazines.

10. Several optional packages are available:
   a) A group of utility programs, including renumber, cross reference, and a program to produce special execute only system disks. You can compile one or more programs to one of these system disks and then run the program without loading Advan BASIC. This means you can give or sell programs to people who don't have Advan BASIC.

   b) A screen design program that lets you design a display which uses several different graphics modes, and even allows you to design alternate character sets. What's more, there are special named subroutines on this disk which you can append to your program. They give you commands for horizontal and vertical fine scrolling and a special plot command for plotting data to these custom displays.

1. GETTING STARTED

Preparation

You will find two Master disks for Advan BASIC in the center of the manual. You should put one away in a safe place; it's your back up disk. You will use the other one to bring up Advan BASIC. Make sure that the write protect tabs are in place on the Master disks so that they can't be erased accidentally.

Before inserting the Master disk the computer should be off and the disk drive on. Also the disk drive busy light should be off. Insert the Advan Master disk and then turn on the computer. After about 35 seconds you will see the message

```
Advan BASIC
(C)Copyright 1985 William Graziano
All Rights Reserved
Ready
```

Advan BASIC is now waiting for you to enter a program. Type in the following lines:

```
10 PRINT "HELLO"
20 END
```

Of course you will need to press the RETURN key at the end of each line. You can use the standard ATARI editor keys to correct errors (see the manual that came with your computer for a description of how these keys work). To make sure everything has been entered correctly, type LIST or L (not L.) and then the RETURN key. This command lists your program and then prints Ready. Now type RUN. Because Advan is a compiled BASIC, the RUN command actually does four things:

1. It compiles your program.

2. If you have an XL or XE computer, it moves about 14 K bytes of the BASIC into high memory. This increases the amount of memory available to the program.

3. It executes the compiled code.

4. It returns the BASIC to its normal location.

If you do not have an XL or XE computer, Advan will skip steps 2 and 4.

You will notice a blanking of the screen at the start and, if you have an XL or XE, at the end of the RUN. Blanking the screen increases execution speed, reducing compile time by about 30%. Advan BASIC has special commands to blank and restore screen display. You can use them to speed up the execution of your program (see Ch. 7). At the completion of RUN, you should see on the screen:

```
HELLO
Ready
```

Entering Programs

Here is a list of facts to keep in mind when entering programs:

1. Line numbers must not exceed 32767.

2. Since long names are used in Advan BASIC, you must use spaces between key words and variable names. For example, 10 FORT=1 TO 3 will give a syntax error. There must be a space between the FOR and T. Normally if you use more than the number of spaces needed, the system will delete the extra spaces. Of course spaces in string expressions (inside quotes) are not affected. Also, spaces at the start of a line are retained. This allows you to indent programs for readability.

3. Multiple statements can be entered on a line; the colon symbol is used to separate statements:

```
10 A=B: PRINT A
```

4. After you enter a line the compiler will check the line for syntax errors. If it finds an error it will print out an error message and then print out the line up to the point where it identified the error. In many cases this will help locate the mistake. If the line is long you might actually have typed a few characters of the next line before the error is detected. The system will make a short buzzing sound to alert you that an error has occurred. When it displays the error message it will overwrite any characters you have typed. Normally you'll not lose more than a few characters. Appendix C has a list of the error messages.

5. If you want to stop what the system is doing, press the BREAK key. For example, if a program gets into an infinite loop, press the BREAK key to return to the BASIC. You can also use the BREAK key to stop a listing or a compile. Note that the BREAK key will not stop a disk save or load.

6. Like the ATARI BASIC, Advan BASIC uses CONTROL 1 to stop and restart the display. Press the CONTROL key together with the 1(one) key to halt the display. Press the two keys a second time to restart the display. Be sure to press the CONTROL key and then the 1 key. The 1 key should be released before the CONTROL key.

7. As mentioned above, you can use the ATARI edit keys to correct or modify program lines. And just like ATARI BASIC, after you have corrected a program line you must move the cursor onto the line and then press the RETURN key.

8. Do not forget to type NEW before entering a program.

9. WARNING: Do not use the RESET key, or you will lose your program and have to reload the BASIC.

Disk Commands

Before you can use a disk you must format it. You can use ATARI DOS 2.0 to format disks, but it is usually easier to use the Advan format program. Of course you will need a blank disk or a disk whose data you don't want to save. With the Master disk still in drive 1, type EXEC FORMAT.COD. This will load the format program into the computer. When the program is ready

to format, it will give the following message:

Insert the disk to be formatted
and then enter the drive number.

After this message appears remove the Master disk and insert the disk to be formatted into drive 1. Next type 1, but don't press RETURN. You will see the following message:

Type Y to format drive 1 disk
Type N to abort

This gives you a last chance to make sure you have the right disk in the drive. Type Y without pressing RETURN. The formatting will take about 30 seconds and then you will be asked if you want to format another disk. Type N without pressing RETURN. The system should respond with Ready. If you have a multiple disk drive system you can leave the Master Disk in drive 1 and insert the disk to be formatted in drive 2. Then type 2 and the disk will be formatted.

Advan BASIC has several commands to control the transfer of programs to and from disks. The three main ones are LOAD, SAVE, and DIR. DIR is used to list the names of the files on a disk. It is a built-in command, so you can use it without harming any program you may have in the computer. To check the disk you just formatted, type DIR. If you have a multiple drive system, type DIR D2:. Don't forget the RETURN key. You should get the message:

707 Free Sectors

Now let's do a sample program so you can see how to save and load. You will have to enter a new program, because the EXEC FORMAT.COD command erased your former program. First type NEW. Then type in the following:

```
10  REM TEST PROGRAM
20  SUM=0
30  SUM1=0
40  FOR T=1 TO 10
50   SUM=SUM+1
60   FOR Y=1 TO 1000
70    SUM1=SUM1+1
80   NEXT Y
90  NEXT T
100 PRINT SUM,SUM1
110 END
```

List the program to make sure you entered it correctly. Correct any errors and type RUN. While ATARI BASIC executes this program in about 56 seconds, Advan BASIC takes less than 16 seconds. If you had used integers, it would take only 6 seconds. I'll discuss integers in the next chapter. The program should print 10 10000 followed by Ready.

To save the program on your disk, type SAVE TEST.BAS if the formatted disk is in drive 1. Type SAVE D2:TEST.BAS if the formatted disk is in drive 2. The screen will be blanked to maximize the speed of the transfer. After a few seconds the display will return and the system will print Ready. To check the disk, type DIR (or DIR D2: if the disk is in drive 2). You should

get:

```
  TEST    .BAS 2
705 Free Sectors
Ready
```

The 2 after TEST.BAS tells you that the program is occupying only 2 sectors on the disk. Neither SAVE nor DIR affects the program in memory. To verify this, type L or LIST. You should find the program unchanged.

Now type LOAD TEST.BAS (or LOAD D2:TEST.BAS if using drive 2). The LOAD command first erases any program in memory and then brings in the program whose name is specified. In this case it will load TEST.BAS. The load will take a few seconds and the screen will be blanked to minimize load time. After the load is completed, the display is restored and Ready appears on the screen.

LIST the program again to make sure it has been loaded correctly. You can now make changes in the program and then save it using the same name or a new one. If you use the same name the previous file will be erased.

If you just want to RUN the program you don't need to load it. Type RUN TEST.BAS (or RUN D2:TEST.BAS if using drive 2). This will erase any program in memory, load the specified program, compile it, and execute it. After a few seconds it should again print 10 10000 followed by Ready.

2. VARIABLE TYPES

The Advantages of Integers

Advan BASIC can store numbers in either integer or real (floating point) form. Integers have no fractional part, and their values range from a maximum of 32767 to a minimum of -32768. Real numbers vary from 10^99 to -10^99; 9 or 10 significant digits will be kept, depending on the number.

Speed and space are the reasons to use integers. Calculations using integers are about 3 times faster than those using real numbers. Also each integer uses two memory bytes, while a real number needs 6 bytes. So if you have a big program and want it to fit into the computer and run as fast as possible (and who doesn't), you should use integers wherever you can.

Names of real and integer variables can be as many characters long as you desire. Only capital letters, numbers, and decimal points may be used in the name. Integer variables must end with a % symbol. Here are some examples of valid names:

Integer Variables	Real Variables
A%	A
BETA.ALPHA%	BETA.ALPHA
BETA.ALPHA1%	BETA.ALPHA1
C123%	C123
BETA1.GAMMA%	BETA1.GAMMA

Unlike some BASICs, all the characters in Advan BASIC variable names are significant. Thus, BETA.ALPHA1 is not the same variable as BETA.ALPHA or as BETA.ALPHA1%. Numerical constants are set to integer form if they end in a % sign and to real form if they do not:

Integer Constants	Real Constants
1%	1.32
-5327%	5
0%	-17E8
31765%	5.27E-3

In Advan BASIC you can use all the standard arithmetic operators (+, -, *, /, ^) with either integer or real variables. In the integer mode the divide operator (/) discards the remainder (it does not round). Thus 9%/5% will yield one. For integer variables there is also a MOD operator. 9% MOD 5% will divide 9 by 5 and return the remainder (in this case, 4).

You can mix integer variables and constants with real variables and constants without causing any errors, but you will reduce execution speed. If a computation must be performed involving a real number and an integer, the integer is converted to a real number before the calculation is carried out.

If an integer variable is set equal to a real number, the real number is first rounded and then converted to an integer. For example in A%=5.7, A%

will be set equal to 6. In A%=B+2%, suppose B equals 9.8. First, the 2% is converted to a real number and then added to 9.8. This gives 11.8. Then 11.8 is rounded to 12 and A% is set equal to 12. Note the difference between this and the divide command. There the remainder was discarded; here the answer is rounded.

Appendix B is a list of words reserved by Advan BASIC. If you use a reserved word for a variable name, you will get a syntax error. Also, unless you are defining a function routine, real and integer variable names must not start with FN.

Strings

Advan BASIC handles strings quite differently from ATARI BASIC. In ATARI the length of each string must be given in a dimension statement and there are no string arrays. This makes strings harder to use. In Advan BASIC, however, you do not need to specify the length of strings. You use dimension statements only to set up arrays (including strings). The maximum string length is 256.

String variable names must end with a $ symbol. You may combine (concatenate) strings using the plus operator. For example, the following program will print ABCDEFG:

```
10 A$="ABC"
20 B$="DEF"
30 GB52$="G"
40 C$=A$+B$+GB52$
50 PRINT C$
```

There are a number of useful built-in functions to help you work with strings, such as LEFT, RIGHT, MID, INSTR, STR$, LEN, CHR$, ASC, and others. See Chapter 10 for additional material.

You will get a syntax error if you use a string name reserved by Advan BASIC (See Appendix B.) Also, unless you are defining a string function, string variable names must not start with FN.

Arrays

The DIM statement indicates that a variable is the name of an array. The number of subscripts can vary from 1 to 64. For example:

```
10 DIM A%(5,3),B$(6,8,2),A(5)
```

This line indicates that A%, B$, and A are arrays. The numbers in parentheses are the maximum values of the subscripts. The minimum value of a subscript is always zero. The DIM statement must always be in a line which precedes the use of the variable.

Special Note: In many BASICs you do not have to dimension an array if the maximum value of a subscript is 10 or less. This will not work with Advan BASIC. All arrays must be dimensioned. Also, all variables and all arrays are zeroed at the start of a program execution.

Integerexpression, Stringexpression, Realexpression

Throughout this manual, commands are usually described by first giving the general format. For example, the command GRAPHICS is used to set the screen display mode. If A% equals 4% and B% equals 2%, then each of the following statements will set the display to mode 8.

```
10 GRAPHICS 8%
10 GRAPHICS A%+4%
10 GRAPHICS B%+6%
10 GRAPHICS A%*B%
```

Rather than give a series of examples, the general format is given like this:

GRAPHICS integerexpression

Integerexpression is simply a short way of describing an expression which yields an integer. In the same way, the general term stringexpression is used in some formats. Here are some examples of stringexpressions:

```
"EIGHT"
A$
A$+"ABC".
```

An expression with only real numbers and real variables or one with a mixture of real and integer will be called a realexpression. Remember that in a mixed expression the integers are converted to real numbers.

If the format for a command specifies an integerexpression, in almost all cases a realexpression can be used in its place. The system will convert the real number to an integer; however, this will slow down the execution of the command to some extent. For example, the following statements involving the GRAPHICS command are acceptable even though the expressions following the command are not integerexpressions:

```
10 GRAPHICS 8
10 GRAPHICS A+4%
10 GRAPHICS B+6
10 GRAPHICS A*B.
```

INPUT

You can use the INPUT command with or without a prompting statement. For example, the command INPUT A%,B$ will print a question mark on the screen and then let the user type in data for A% and B$. If you want to give the user some instruction about what to enter, you can use the INPUT command with a prompting message:

```
10  INPUT "Enter date in form DD-MMM-YY "T$
```

The above line will print---Enter date in form DD-MMM-YY---and then let the user type in the date. Note that a question mark is not printed; if you want one, you must put it in the message. The command INPUT ""T% will print neither a question mark nor a prompting message.

INPUTLINE

The INPUT command is limited to strings without commas, because commas are already used to separate data items. For example, if you give the command INPUT A$,B$ and the user enters SMITH, JOHN then A$ will be set to SMITH and B$ to JOHN. Or if you give the command INPUT A$ and the user enters SMITH, JOHN then A$ will be set to SMITH and JOHN will be ignored. The INPUTLINE command solves this problem. It brings in all of the data entered by the user (except the carriage return) and sets it equal to a string. For example:

```
10  INPUTLINE A$
```

If the user types SMITH, JOHN in response to this request for data, A$ is set equal to SMITH, JOHN. Note that you must always input the data into a single string variable. All the following commands will give syntax errors: INPUTLINE A INPUTLINE A$,B$ INPUTLINE A$,B%.

A promptimg message may be used with INPUTLINE:

```
INPUTLINE "ENTER NAME "T$
```

READ and DATA

Advan BASIC READ and DATA commands work about the same way as they do in ATARI and most other BASICs. Note that string information in DATA statements should not be enclosed in quotation marks:

```
10  READ A%,A$,B
20  DATA 5,ABC,2.7
30  PRINT A%,A$,B
RUN
5   ABC   2.7
```

One problem in data statements is how to get a comma into a string; the same problem as with the INPUT command. Since commas are used to separate data items, if you want to use one or more commas in a string you need to do something special. In Advan BASIC you can enter a comma in INVERSE mode.

When the system reads a string from a DATA statement, it converts an INVERSE comma to a regular comma. To enter an INVERSE comma, hit the INVERSE key and then the comma. Hit the INVERSE key one more time to get out of the INVERSE mode and back to normal mode.

If you should want to use an actual inverse comma as part of a data string, you've got a real problem. Let's hope you never want to do this.

Special Note: Unlike ATARI BASIC, Advan BASIC allows you to INPUT, READ, or INPUTLINE to an array variable:

```
10 INPUT A%(5%)
```

GOTO and GOSUB

Advan BASIC has the standard BASIC commands, GOTO and GOSUB. Also available are named subroutines with arguments. They can simplify the writing of programs and make them easier for someone else to understand. That "someone" could be you if you need to modify a program you wrote some time ago. Chapter 8 is devoted to functions and subroutines.

IF THEN ELSE

The IF THEN ELSE command lets you take different actions depending upon whether or not a specified condition is true or false:

```
10 IF DATE$="1 JAN" THEN PRINT "1st" ELSE PRINT "Not 1st"
```

The IF command is always followed by a conditional expression, which may contain integers, real, or string expressions. Conditions can also be combined with the AND and OR commands:

```
10 IF DAY%=31% AND MONTH$="DEC" THEN PRINT "IT'S NEW YEAR'S EVE"
```

If you are using both AND's and OR's, the AND's will always be executed first unless you use parentheses. In the next example, the OR commands in parentheses are executed before the AND.

```
10 IF (T%<3% OR Y%<>2%) AND (Z%>5% OR J%<=1%) THEN 100
```

You can have a line number or one or more statements after THEN. ELSE is optional; if used, it can be followed by a line number or by one or more statements. IF THEN ELSE must fit on one BASIC line (no more than 3 screen lines).

IF DO ELSE ENDIF

IF DO ELSE ENDIF is a multi-line version of IF THEN ELSE. Just as in IF THEN, a conditional expression must always follow IF. DO replaces THEN and tells the compiler that it's a multi-line IF. In IF THEN, the end of the BASIC line is also the end of the IF. In the DO form, the IF continues until the ENDIF command is reached. ELSE is optional:

```
10  IF MONTH$="JAN" AND DAY%=1% DO
20     YEAREXPENSES=0
30     YEARINCOME=0
40     YEARTAX=0
50  ENDIF
```

In the above example, if it's Jan 1st, lines 20, 30, and 40 are executed and the three variables zeroed. If it's not Jan 1st, the program will skip lines 20, 30, and 40. Here is another example:

```
10 IF INCOME >20000 DO
20   TAX=0.2*(INCOME-DEPENDENTS*1000-4300)
30   WITHOLDING=TAX/12
40 ELSE
50   TAX=0.1*(INCOME-DEPENDENTS*1000-2000)
60   WITHOLDING=0
70 ENDIF
```

If INCOME is greater than 20000, lines 20 and 30 are executed and the program skips to the statement after the ENDIF. If INCOME is less than or equal to 20000, lines 20 and 30 are skipped and lines 50 and 60 are executed.

ON GOTO

While the IF command gives you two options, ON GOTO gives you many options. The ON must be followed by a real or integer expression. If it is real it will be converted to an integer (rounded). This expression is followed by GOTO and a list of linenumbers:

```
10  ON T%+1% GOTO 100,50,200,400
```

You can have as many line numbers following the GOTO as will fit on the BASIC line. The system evaluates the numerical expression. If it equals 1, the program executes a GOTO to the first line number in the list. If it's a 2, the GOTO is to the second line number, etc. In the above example, if T%=3, the system will evaluate T%+1% and get 4. The program will then GOTO line 400. Note that you cannot use calculated line numbers, such as ON T% GOTO 100+10,20.

ON GOSUB

ON GOSUB works the same way as ON GOTO, except that a GOSUB to a line is executed instead of a GOTO. When the subroutine is finished, the program returns to the command immediately following ON GOSUB:

```
10  ON T%/3%+1% GOSUB 90,10000,100,200
```

CASE

The CASE command provides another way to do a multi-way choice. The following diagram shows the general format of the CASE command:

```
CASE condition
       (statements)
  & condition
       (statements)
  & condition
       (statements)
     .
     .
     .
  CASE ELSE
       (statements)
CASE END
```

A conditional expression always follows the CASE command. If the condition is true, the statement(s) between it and the & symbol are executed and the program skips down to the statement immediately following CASE END.

If the first condition is false, the program jumps to the & symbol and checks the condition following it. If that condition is true, the statement(s) between it and the next & are executed, and the program jumps to the statement immediately following CASE END. If the second condition is false, the program skips to the next & symbol, and repeats the process. In the above format, only two & conditions are shown. You may use as many as you want.

The CASE ELSE and statement(s) following it are optional. If all the previous conditional expressions are false, the program will execute the statement(s) immediately following CASE ELSE (if it is present); if it is not present, the program jumps to the statement immediately following CASE END. The following example shows how to use the CASE command to determine the number of days in a specified month:

```
10  CASE MONTH%=2%
20     DAYS%=28%
30  & MONTH%=4% OR MONTH%=6% OR MONTH%=9% OR MONTH%=11%
40     DAYS%=30%
50  CASE ELSE
60     DAYS%=31%
70  CASE END
```

Advanced Topics: (Programmers with limited experience should probably move on to the next chapter.)

IF THEN, IF DO, and CASE commands can be nested. For instance, you can use an IF THEN within an IF THEN, an IF DO or a CASE. The one requirement for IF THEN is that everything must fit into one line. Here is an example of an IF THEN within an IF THEN:

```
10  IF T%>2% THEN IF Y%>2% THEN 100 ELSE 200
```

Note that if there are two IF'S and only one ELSE, the ELSE goes with the nearest preceding IF. *In the above example, ELSE goes with the IF Y%>2%.* If there had been another ELSE at the end of the line, however, you would have gotten a syntax error. An ELSE may refer only to the nearest preceding IF.

If T% is less than or equal to 2%, the program goes to the next line. If T% is greater than 2% and Y% is greater than 2%, the program goes to line 100. If T% is greater than 2%, but Y% is not greater than 2%, the program goes to 200. Here is another example:

```
10  IF T%>2% THEN 100 ELSE IF Y%>2% THEN 200 ELSE 300
```

If T% is greater than 2%, the program goes to 100. If T% is not greater than 2%, but Y% is greater than 2%, the program goes to 200. If neither T% nor Y% are greater than 2%, the program goes to 300.

When the program evaluates a conditional expression, it returns an integer equal to one if the expression is true and zero if it is false. You can sometimes use this fact to your advantage. Suppose you want to add 5 to T%

if Y% is greater than 2%. The following statement will achieve this:

```
10  T%=T%+(Y%>2%)*5%
```

If Y% is greater than 2%, the expression in parentheses will be set equal to 1 and multiplied by 5. If Y% is not greater than 2%, the expression is set to zero and T% will still equal T%. Consider the following example:

```
10 IF T% THEN 100
```

If T% is non-zero (plus or minus), the program jumps to line 100, since Advan BASIC assumes any non-zero number to mean true. If T% equals zero, the system considers the expression to be false and goes to the next line.

5. LOOPS

FOR NEXT STEP

FOR NEXT is the main looping technique in most BASICs. Advan BASIC allows two kinds of FOR loops, integer and real. The name of the loop variable determines the loop type. The lower limit, upper limit, and step can be integer or real, whether or not the FOR loop is integer or real. If there are differences in type, the system will convert so that the lower limit, upper limit and step are of the same type as the loop variable.

In some BASICs you do not need a variable name with the NEXT statement. In Advan BASIC you must include the variable name with each NEXT, and the system will check to make sure that corresponding FOR and NEXT statements have the same variable name. As in most BASICs, FOR NEXT loops may be nested. Here a real loop is nested inside an integer loop:

```
10  FOR T%=1% TO 5%
20     FOR Y=1 TO 3
30         PRINT T%,Y
40     NEXT Y
50  NEXT T%
```

And here an integer loop is nested inside a real loop:

```
10  FOR T=5% TO 1% STEP -1
20     FOR Y%=1 TO 3 STEP 2
30        PRINT T,Y%
40     NEXT Y%
50  NEXT T
```

The 5% and 1% in line 10 will be converted to real numbers, and the 1, 3, and 2 of line 20 will be converted to integers.

If you are writing programs to move data around inside the computer, you generally want to work with integers because they are faster. Remember, the largest value of an integer is 32767. Advan BASIC however, will let you use FOR loops and POKEs with integers up to 65535% It simply stores them as negative numbers. Consider the following program, which stores zeroes in the bottom line of a mode 0 text display (memory locations 40920 to 40959):

```
10  FOR T%=40920% TO 40959%
20     POKE T%,0%
30  NEXT T%
```

As far as the program is concerned, it is executing a loop from -24617% (same as 40920%) to -24577% (same as 40959%).

WHILE WEND

If you know how many times you are going to repeat a loop, FOR NEXT statements work very well; if you do not, FOR NEXT is not so easy to use. For example, suppose you write a program in which the user enters numbers to be summed, with -1 entered after the last number. Here you have no idea how many numbers will be entered. A WHILE WEND or REPEAT UNTIL loop,

however, allows you to write the program easily. The following shows the standard form for a WHILE WEND loop; WEND stands for WHILE END:

```
WHILE condition
    (statements)
WEND
```

When the program reaches the WHILE command, it evaluates the condition. If false, it jumps to the statement after the WEND. If true, it executes the statements up to the WEND, jumps back to the WHILE, and again evaluates the condition. As long as the condition is true, the program will stay in the loop. The following program uses WHILE loops to solve the problem posed above (input and sum numbers until -1 is entered):

```
10 SUM=0
20 INPUT NUMBER
30 WHILE NUMBER<>-1
40    SUM=SUM+NUMBER
50    INPUT NUMBER
60 WEND
70 PRINT SUM
```

Note that you have to use two INPUT NUMBER commands; once so that the loop can get started, and again in the main body of the loop. This is not unusual for WHILE loops. One of the more common errors with WHILE loops is forgetting to do something to change the condition. For example, if we had forgotten the INPUT on line 50, then NUMBER would never be changed and we would have had an infinite loop. Because of their upper and lower limits, FOR loops do not have that problem. The following is an alternate form of the WHILE loop:

```
WHILE condition DO
    (statements)
WEND
```

Both forms work in the same way. You might prefer the appearance of the latter.

REPEAT UNTIL

The format is:

```
REPEAT
  (statements)
UNTIL condition
```

When the program reaches the REPEAT, it continues past it and executes the following statement(s). When it reaches UNTIL, it evaluates the condition. If true, it leaves the loop and goes to the statement following UNTIL. If false, it jumps back to the statement after REPEAT and goes through the loop again.

REPEAT UNTIL serves about the same role as the WHILE loop, except that the condition is at the end of the loop instead of at the beginning. Therefore, the loop in a REPEAT is always executed at least once, while the statements in a WHILE loop are not executed at all if the initial condition is false.

There is another difference between the REPEAT and WHILE loops. The program stays in a WHILE loop until the condition becomes false; it stays in a REPEAT loop until the condition becomes true. The following shows another way to input and sum numbers until -1 is input:

```
10  SUM=0:NUMBER=0
20  REPEAT
30     SUM=SUM+NUMBER
40     INPUT NUMBER
50  UNTIL NUMBER=-1
60  PRINT SUM
```

6. DISK INPUT AND OUTPUT

You already worked with disk files when you saved and loaded programs in Chapter. 1. This chapter deals with disk data files. Advan BASIC uses essentially the same file structure as ATARI DOS 2.0; you can use that DOS to copy files, delete files, rename files, etc. The Advan disk commands, however, differ from those used by ATARI BASIC.

OPEN

Before you can work with any disk file, you must first give an OPEN command. Here is the general format:

OPEN stringexpression,integerexpression,stringexpression

The stringexpression immediately following the OPEN command must be I, O, A, or R.

I--inputs data only from the file to the computer

O--outputs data only from the computer to the file. If a file wth the same name already exists on the disk, using an O will destroy that file. If you want to add to a file, use A (append).

A--sends data from the computer and adds it to the file specified in the file name.

R--opens a file in the so-called random mode. I, O, and A are essentially sequential modes. They transfer data only in sequence, from the beginning to the end. In the R mode, however, you can get and put data anywhere in the file. Unfortunately, the ATARI disk format makes the R mode somewhat difficult to use; more on that later in the chapter.

The integerexpression is used to assign a channel number to the file. This number must be 0, 1, 2, or 3. All the other Advan BASIC commands use this channel number when working with the file. Note that two files may not have the same file number simultaneously.

The last stringexpression gives the filename, which has three parts:

1. A 'D' followed by the disk number (1 to 4) that the file is on or will be on, and then a colon. If the file is on disk 1, you can omit this part.

2. The main file name, which must be 1 to 8 characters long.

3. An optional filename extension, which is a period followed by 1 to 3 characters.

The following are valid file names: ADDFIL ADDFIL.DAT D3:BETA

D2:NEWFILE1.C13

The following are legal OPEN statements: OPEN "I",0%,"ALPHA.DAT"

OPEN "O",1%,"D2:B" OPEN "R",3%,"C.DT" OPEN "A",2%,"D3:BETAONEA.FIL"

CLOSE

After you have finished working with a file, you need to give a CLOSE command for it. This frees the channel number so that it can be used with other files. Also, it puts the final pieces of information on the disk if any output was made to the file. If you have written to a disk and do not close the file, you may lose some of the information. 10 CLOSE 1% will close the file that was opened on channel 1.

PUT

Use the PUT command to write information into a file. The file must have been opened in the O, A, or R mode. Also, the disk must not have a write protect tab on it, and the file must not be locked (see LOCK command in reference manual). The general format for a PUT command is

PUT integerexpression,variablename

The integerexpression is the channel number and must be 0, 1, 2, or 3. The variable is what is being stored on the disk. It can be an element of an array:

```
10  PUT O%,N%(2%)
```

In the following example, 100 names and associated salaries are input from the keyboard and saved to a file:

```
50   OPEN "O",1%,"SALARY.DAT"
100  FOR NUMBER%=1% TO 100%
110    INPUTLINE NAME$
120    INPUT SALARY
130    PUT 1%,NAME$
140    PUT 1%,SALARY
150  NEXT NUMBER%
160  CLOSE 1%
```

I should mention that the maximum length string which you can store on a disk is 255 bytes (not 256).

GET

Use the GET command to read information from a file. The file must have been opened in the I or R mode. The general format for the GET command is

GET integerexpression,variablename

The integerexpression is the channel number and must be 0, 1, 2, or 3. The variable will contain what is read from the disk; it can be an element of an array.

In the following example, the 100 names and associated salaries stored in the above program are read back from the file into two arrays, and then printed:

```
50  OPEN "I",0%,"SALARY.DAT"
60  DIM NAME$(100),SALARY(100)
100 FOR NUMBER%=1% TO 100%
110   GET 0%,NAME$(NUMBER%)
120   GET 0%,SALARY(NUMBER%)
130 NEXT NUMBER%
140 CLOSE 0%
200 FOR NUMBER%=1% TO 100%
210    PRINT NAME$(NUMBER%),SALARY(NUMBER%)
220 NEXT NUMBER%
```

Suppose you want to change one of the salaries? You can use the R mode to read through the file searching for the name, and then change the salary. The following program will do this:

```
10  PRINT "ENTER NAME OF PERSON"
20  INPUTLINE "WHOSE SALARY IS TO BE CHANGED " T$
30  INPUT "ENTER NEW SALARY " T
50  OPEN "R",2%,"SALARY.DAT"
60  FOR T%=1% TO 100%
70     GET 2%,NAME$
80     IF NAME$=T$ THEN PUT 2%,T: GOTO 110 ELSE GET 2%,SALARY
100 NEXT T%
110 CLOSE 2%
```

EOF

You use the EOF command to check whether or not you have read all of the data from a file; that is, whether you are at the end of the file. Use it when you do not know the length of a file. For example:

```
10  T%=EOF(1%)
```

If you are at the end of the file for channel 1, EOF will be set to the integer 1 and, thus, T% will be set to 1. If there is still data to be read (that is, you are not at the end of the file), T% will be set to 0. The following segment reads the names and salaries from the file, but does not assume that the number of items is known:

```
50  OPEN "I",2%,"SALARY.DAT"
100 WHILE EOF(2%)<>1%
110    GET 2%,NAME$
120    GET 2%,SALARY
130    PRINT NAME$,SALARY
140 WEND
150 CLOSE 2%
```

The remaining material in this chapter is somewhat more difficult. Programmers with limited experience should probably skip to the next chapter.

NOTE and POINT

The NOTE command is used to remember where a given piece of data is located in the file. The POINT command is used to return to that location. The format of the two commands is similar:

NOTE channelnumber,integer variable,integer variable

POINT channelnumber,integerexpression,integerexpression

Consider the following program segment. The NOTE command on line 10 stores the sector number of the file opened on channel 1 in the variable, SECTOR%. The current position in the sector is stored in SECTORPOS%:

```
10 NOTE 1%,SECTOR%,SECTORPOS%
.
.
.
90 POINT 1%,SECTOR%,SECTORPOS%
```

Suppose several disk operations have occurred between lines 10 and 90. When line 90 is executed, the disk operating system returns to the file position it was at when line 10 was executed.

```
10  OPEN "O",1%,"TEST"
20  FOR T%=1% TO 5%
30      IF T%=3% THEN NOTE 1%,SECTOR%,SECTORPOS%
40      T1%=T%+5%:PUT 1%,T1%
50  NEXT T%
60  CLOSE 1%
100 OPEN "R",1%,"TEST"
110 POINT 1%,SECTOR%,SECTORPOS%
120 GET 1%,TEST%
130 PRINT TEST%
```

The above program creates the file named "TEST" on disk 1, and stores in it the numbers 6, 7, 8, 9, and 10. SECTOR% and SECTORPOS% will contain the information on where the third number of the file is located. Line 100 opens the file in the random mode so that we can get just the third number. The POINT command positions us at the start of the third number, and the following command gets the number and stores it in TEST%. Line 130 prints '8', the third number of the file.

With POINT and NOTE you can set up a file, and GET and PUT to it without reading or writing the whole file. The trick is to use the NOTE command for, say, every sixteenth file element, and to store the sector and sector position data in an array. Then, when you want to get to an element, you use the data in the array with a POINT command to position as close as possible to the element. Finally, you use GET commands to reach the particular data element. If you have a big file and need to access records quickly, this is one way to do it. You can even store the array in a file, and when you want to work with the main file, you first read the array from the disk. See reference manual section on NOTE for an example.

GET and PUT (alternate form)

There is an alternate form of GET and PUT which allows you to specify the number of bytes to be transferred. For example, GET 1%,A%(0%),50% will read 50 bytes from the file into the array A%. Since an integer is two bytes long this will read values for A%(0%) through A%(24%). This is much faster than getting the integers one at a time. PUT 1%,A(10%),20% will put A%(10%) through A%(19%) into the file. On a GET you need to be careful that the array is large enough to hold all the bytes. For instance, if A%

had been dimensioned only to 20, then GET 1%,A%(0%),50% would crash the program. In an extreme case you could even crash the system.

If you try to get more bytes than are in the file, you will get an end of file error message. An integer representing the number of bytes not transferred will be stored at location 1238 (low order) and 1239. Thus, if you try to get 30 bytes and there are only 10 bytes in the file, location 1238 will be set to 20 and 1239 to zero. Use PEEKW(1238%) to find the number of bytes not transferred.

You can also use this alternate form to GET and PUT one byte. For example, PUT 2%,N%,1% will put a byte of value equal to N% to the file. N% must have a value less than or equal to 255 and greater than or equal to zero. GET 2%,N%,1% will get a byte from the file and place it in the low order byte of N%. Note that the high order part of N% will not automatically be zeroed. In many cases, it is a good idea to set the variable to zero before getting a single byte. This forces the high order byte to zero.

WAIT

The WAIT command forces the computer to pause for a specified time. Its format is

WAIT integerexpression

The value of the integerexpression will tell the computer how long to wait in sixtieths of a second. The first example below will cause a 15/60 second pause:

```
10 WAIT 15%
```

```
10 WAIT T%+2%
```

RTIME

The RTIME command resets the clock to zero:

```
10 RTIME
```

This command is normally used with the TIME function. See Chapter 8--Functions and Subroutines.

OFFDISPLAY and ONDISPLAY

Maintaining the display slows down the CPU. In mode 0, the normal text mode, the system will run about 30% faster with the display off. If you want to go as fast as possible and don't need the display, use the OFFDISPLAY command; the ONDISPLAY command turns it back on:

```
10  OFFDISPLAY
20  SUM%=0%
30  FOR T%=1% TO 30000%
40     SUM%=SUM%+1%
50  NEXT T%
60  ONDISPLAY
70  PRINT SUM%
```

DEG and RAD

When the BASIC is loaded, it is initialized so that angles used with trig functions must be in radians. The DEG command changes the trig functions so that they work with degrees. The RAD command switches the system back to radians.

```
10 DEG
     .
     .
     .
50 RAD
```

POKE and POKEW

POKE is a commonly used command; it stores a byte in a specified memory location. POKEW stores a 16 bit word in a specified memory location. The low order 8 bits are stored at the memory location, and the high order bits are stored at memory location plus one. This is the normal way to store an integer on the ATARI. In the following example, 0 is stored in 40959, and 289 is stored in 40957 and 40958:

```
10 POKE 40959%,0%
20 POKEW 40957%,289%
```

EXG

The EXG command exchanges two strings. Its format is:

```
EXG(stringvariable,stringvariable)
```

```
10 EXG(T$,A$)
```

```
10 EXG(Y1$(5%,2%),C$(1%))
```

In the first example, if A$ equals 'A' and T$ equals 'ZZ' before the EXG command is executed, then A$ will equal 'ZZ' and T$ will equal 'A' after EXG is executed. Note that you cannot use EXG with integers or real variables.

TRAP

Normally, if an error occurs during the execution of a program, the system immediately returns to BASIC. Sometimes this is undesireable. For example, suppose a user enters and misspells a file name; the program cannot find the file and returns to BASIC with a file not found error. Wouldn't it be better to give the unfortunate person another chance? The TRAP command lets you do this. Its format is

```
TRAP linenumber
```

In case of an error, TRAP causes the system to go to the specified linenumber. Memory location 1240 will have the error number. You can use a PEEK command to get it. Appendix B has a list of the error codes. Suppose you come to a section of the program where you would prefer an error to force a return to BASIC. TRAP 0 will cause subsequent errors to do this. Of course, you can always issue another TRAP command and again take over error control.

LOADST and POPST

LOADST lets you save strings or numbers to the stack used by the system. POPST lets you remove strings and numbers from the stack and store them in variables. The formats are

```
LOADST(expression)
```

```
POPST(variablename)
```

For example, suppose you have a subroutine which takes a number and

returns a string equal to the hex value of the number. The question is, how to supply the subroutine with the number, and how to get the string information back from the subroutine. One way is to use the LOADST command to put the number on the stack. The subroutine uses the POPST command to store the number in a variable that it can work with. When the subroutine finishes evaluating the string, it loads the string on the stack and then returns. The main program now uses the POPST command to store the string in a variable:

```
10  LOADST(T%)
20  GOSUB 100
30  POPST(T$)
.
.
.
100 POPST(HEX%)
.
.
.
200 LOADST(HEX$)
210 RETURN
```

8. FUNCTIONS AND NAMED SUBROUTINES

Built-in Functions

Advan BASIC provides a number of built-in functions. In addition, it allows user-defined functions and named subroutines with 0 to 4 arguments. Most of the built-in functions are listed or described in the chapter covering the pertinent topic. The remaining functions are described in this chapter. Table 8-1 lists the functions that deal with real variables; Table 8-2 those dealing primarily with integers.

Table 8-1

ABS(X)	returns the absolute value of X
ATAN(X)	returns the arctan of X
COS(X)	returns the cosine of X
EXP(X)	returns e^X
FINT(X%)	treats X% as an unsigned integer from 0% to 65535% and converts it to a real number
FIX(X,Y%)	rounds X to Y% decimal places.
INT(X)	returns the integer part of X
LOG(X)	returns the natural logarithm of X
RND(X)	returns a random number with a value between 0 and X
SIN(X)	returns the sine of X
SGN(X)	returns 1 if X>0, 0 if X=0, and -1 if X<0
SQR(X)	returns the square root of X
TAN(X)	returns the tangent of X
VAL(X$)	translates a string into the number that it represents

Table 8-2

ABS%(X%)	returns the absolute value of X%
GETKEY	returns the ASCII code for a key that has been pressed, returns 0 if no key has been pressed
PEEK(X%)	returns an integer equal to the value at the memory location X%
PEEKW(X%)	returns an integer equal to the value of the word at memory location X% and X%+1%; (low order 8 bits at X% and high order 8 bits at X%+1%
RND%(X%)	returns a random integer of value greater than 0 and less than or equal to X%. X% must be less than 256%.
STICK(X%)	returns an integer whose value depends upon the position of joystick numbered X% (see reference manual)
STRIG(X%)	returns 0 if the X% joystick firebutton is pressed; otherwise returns 1%
TIME	returns an integer whose value equals the number of sixtieth of seconds since last RTIME command

The reference manual gives additional information on all the functions listed in Tables 8-1 and 8-2.

User-Defined Functions

If there is an expression which you evaluate several times in a program, a user-defined function may be helpful. The format is

```
DEF functionname(variablename,...,variablename)=expression
```

The functionname must start with FN. The variablenames in parentheses are optional; there may be no more than four. Suppose that in several places you need to determine the larger of two integer numbers. The following function will do this:

```
DEF FNLARGER%(X%,Y%)=X%*(X%>Y%)+Y%*(Y%>=X%)
```

If X% is greater than Y%, then X%>Y% yields one and Y%>=X% yields zero; thus the right side equals X%. If Y% is greater than or equal to X%, then the right side equals Y%. The values of X% and Y% are determined when the function is used. Here is an example:

```
10 G%=3%: A%=6%
20 PRINT FNLARGER%(G%*3%,A%)
30 DEF FNLARGER%(X%,Y%)=X%*(X%>Y%)+Y%*(Y%>=X%)
RUN
9
```

At line 20 we use the functionname. X% takes the value of G%*3% (i.e., 9%), and Y% takes the value of A% (i.e., 6%). Since X% is greater than Y%, the function returns the value of 9%. Thus, at line 20, FNLARGER% is set to 9% and this is what is printed.

The variables X% and Y% are called dummy variables. Their values are set by the expressions in the parentheses following the use of the functionname in the program. These values hold, however, only in the definition of the function. If X% and Y% had been used elsewhere in the program, their value would have been unchanged by what happened to X% and Y% in the function. That is, the X% and Y% in the function would not be the same variables as X% and Y% used elsewhere in the program.

Functionnames can be real, integer (ends in %), or string (ends in $). String functions always return string expressions; thus, by definition, the right side of a string function must be a string expression.

Named Subroutines

Named subroutines are similar to ordinary subroutines, except they can have 0 to 4 dummy variables, and they are called in a different way. To call a named subroutine, just use its name; if dummy variables are used, set up expressions for each one. Subroutine names must end with a @ symbol. The command SUB followed by the subroutine name, defines the start of the routine. The command SUBEND defines the end of the named subroutine. If you want to exit from the middle of a named subroutine, you can use a RETURN command. Any number of RETURNS may be present; however, you may use only one SUBEND and it must be the last statement in the subroutine.

```
5   A%=7%
10  PRINTLARGER@(5%,3%)
15  PRINTLARGER@(-3%,A%+1%)
20  SUB PRINTLARGER@(X%,Y%)
30     IF X%>Y% THEN PRINT X% ELSE PRINT Y%
40  SUBEND
RUN
5
8
```

In the above example, the PRINTLARGER@ subroutine is called first at line 10. The subroutine starting at line 20 is executed with X% equal to 5% and Y% equal to 3%; a 5 is printed. The subroutine is again called at line 15, with X% now equal to -3% and Y% equal to A%+1%, that is, 8%. An 8 is printed.

The variables X% and Y% are called dummy variables. Their values are set by the expressions in the parentheses following the use of the subroutine name in the program. These values hold, however, only in the subroutine. If X% and Y% are used elsewhere in the program, their values remain unchanged by what happened to X% and Y% in the subroutine. In effect, the dummy variables X% and Y% in the subroutine would be different from the X% and Y% used outside of the subroutine.

Special note: In most situations the system will automatically convert between real and integer numbers. In user defined functions and named subroutines, however, the argument must be the same in the definition and when the function or subroutine is used. For instance, the following program gives an argument error when compiled. This is because the definition at line 20 shows an integer argument, but a real argument is used at line 10.

```
10 PRINT FNT(A)
20 DEF FNT(B%)=2
```

The system will go around a function or subroutine definition automatically, just as it goes around DATA statements. So you may place functions and subroutines anywhere in the program. Note that the definition of a function or a subroutine does not need to precede its use.

9. MORE ON PRINTING

PRINT and LPRINT

The PRINT command is used to display information on the TV or monitor screen. Note that the system considers a ? to be the same as a PRINT command. Using a ? will cut down on your typing and save space on a display line. LPRINT works the same as PRINT, except that the output is sent to the printer. Like most BASICs, Advan BASIC uses commas and semicolons to control print spacing. See the reference manual if you are not familiar with this.

TAB

Use the TAB function to specifiy the column you want to print in. It can be used with either the PRINT or LPRINT command. For example:

```
PRINT TAB(15%); "HELLO"
```

The far left column is 0, the next is 1, etc. TAB(15%) will shift the print position to column 15, the semicolon will keep it at 15, and the H in HELLO will be printed in column 15.

PRINT USING and LPRINT USING

These commands allow the programmer much greater control of output than do PRINT and LPRINT commands. For example, you can:

1. specify the number of decimal points (the number will be rounded).

2. right justify numbers

3. align decimal points

4. insert $ and/or * before numbers

5. insert trailing or leading minus signs

6. do some string operations

The details of using the command are described in the reference manual. I should remind you, however, that to use these two commands the PUSING.APP file must be appended to your program before it is compiled.

WIDTH

This is a system command (you cannot use it in your program) which sets the printing width. Typing WIDTH 80 causes the system to assume that the printer is set for 80 characters. If you do not use this command, the system assumes a printer width of 75 characters. If you want to set the printer width in your program, you must POKE the new width into memory location 1251. For example, POKE(1251%,80%) sets the printer width to 80.

10. MORE ON STRINGS

String Functions

There are a number of Advan BASIC functions which are helpful when working with strings. Each is described here briefly and in more detail in the reference manual.

CHR$(X%) generates a one character string; the ASCII code for this character is given by the value of X%.

CHRW$(X%) generates a two character string; the ASCII code for the first character is X% MOD 256%. The ASCII code for the second character is X%/256%.

INSTR(X%,A$,B$) searches A$ to see if B$ is included in it. The search starts at the X% character position of A$. If no match is found, it returns an integer equal to 0%; otherwise, it returns an integer equal to the character position in A$ at which the match was found.

INSTR1(A$,X%,Y%) searches A$ to see if the character whose ASCII code equals Y% is present. The search starts at the X% character position of A$. If no match is found, an integer equal to 0% is returned; otherwise, it returns an integer equal to the position in A$ at which the match occurred.

LEFT(A$,X%) returns a string equal to the first X% characters of A$.

LEN(A$) returns an integer equal to the length of A$.

MID(A$,X%,Y%) returns a string of length Y% composed of the characters of A$ from position X% to X%+Y%-1%.

NUM$(X%) returns a string representing the value of the integer X%

RIGHT(A$,X%) returns a string composed of the characters of A$ from position X% to the end of the string.

STRING(X%,Y%) returns a string of length X%, all of whose characters have the ASCII code equal to Y%.

STR$(X) returns a string representing the value of the real number X.

In additon to these functions, there are two Advan BASIC commands designed to work with strings:

INSERTB

```
10 INSERTB(A$,X%,Y%)
```

This command inserts a character into A$ at the X% position. The ASCII code of the character will be Y%.

INSERTW

```
10 INSERTW(A$,X%,Y%)
```

This command inserts two characters into A$; one at the X% position and the other at X%+1%. The ASCII code of the character at the X% position is Y% MOD 256%. The ASCII code of the character at the X%+1% position is Y%/256%.

11. MORE ON SYSTEM COMMANDS

SAVEC and EXEC

Here is how to run a regularly used program without having to compile it first each time. Use the LOAD command to bring the program into the computer and then type COMPILE. After the COMPILE is completed, the system will respond with Ready. Now type SAVEC and follow this with a space and filename. This will save the compiled code on the disk. When you want to run the program, type EXEC followed by the filename of the compiled code.

The following example shows how to compile and save the code for a program named ALPHA.BAS:

```
LOAD ALPHA.BAS
Ready
COMPILE
Ready
SAVEC ALPHA.COD
Ready
```

Next time you want to run the program, type EXEC ALPHA.COD.

Compiling and Executing Long Programs

If you do enough programming in Advan BASIC, you will eventually write a program which is too long for the normal RUN command, and you will get a MEMORY EXCEEDED message. This means that either there isn't enough room for the compile, or there isn't enough room for the program and its data during execution. Take heart; in many cases you can still run the program. It just takes a slightly different technique.

When you type RUN, Advan BASIC will keep both the compiled code and the program in memory. This is very convenient if you want to modify the program, because it's right in memory. You simply make the changes and then run it again; however, it does limit the size of a program you can run.

If you type RUN 1, the system will delete each line of the program as the line is compiled. At the end of the compile, the program will have been erased. This lets you run much larger programs. Be sure to save the program before you try this, or you will be very sad indeed. You can also use this technique with programs you run directly from a disk. For example:

```
RUN D2:ALPHA.BAS 1
```

This command loads the program from disk 2, compiles it, and then executes it. The program lines are deleted as they are compiled. You can also use this option with the COMPILE command:

```
COMPILE 1
```

This command compiles the program in memory and deletes each line after it has been compiled.

If you can now compile, but still cannot execute, you can squeeze out a little more memory by typing RUN 3. If you do not have an XL or XE computer, this will gain about 17K of space. The system thinks of this as a 1 plus a 2. The 1 tells it to delete the program; the 2 tells it to remove the BASIC. On an XL most of the BASIC is put in special high RAM before execution, and the gain is rather small (about 3K). If it allows the program to execute, however, don't knock it. You can also use the 3 option when running programs from a disk. The following command is equivalent to loading ALPHA.BAS from disk 1 and then giving a RUN 3.

```
RUN ALPHA.BAS 3
```

If you can compile using a RUN 3, but still cannot execute, you have one of three possibilities:

1. rewrite the program
2. use a different BASIC, computer, or language
3. use the optional Advan BASIC optimizing compiler

What if you cannot even compile with the COMPILE 1 option? Then you will need to do a disk to disk compile. This will bring in the parts of the program only as they are needed, and will put the output code back on the disk. With it, you can compile quite long programs. Note that your program name must not end in .COD or .WRK, or you will lose it. There are two formats possible:

(1) COMPILE ALPHA.BAS

This command compiles the program named ALPHA.BAS on disk 1 and stores the compiled code on drive 1 under the name ALPHA.COD. A file named ALPHA.WRK will be created and then erased at the end of the compile.

(2) COMPILE ALPHA.BAS/D2:GAMMA.OO2

This command compiles the program named ALPHA.BAS on disk 1 and stores the compiled code on drive 2 with the name GAMMA.OO2. The / symbol tells the system that you want to specify the location and name of the compiled code. After the disk to disk compile is completed, you can execute the compiled code with the EXEC command. The following command executes the code generated by the previous command:

```
EXEC D2:GAMMA.OO2
```

If you add a space and a 2 at the end of either of the above options, the BASIC will be removed when the program is executed:

```
COMPILE ALPHA.BAS/D2:GAMMA.OO2 2
```

If you remove the BASIC to execute a program, you must restore the BASIC at the end of the program. The system will print the message 'INSERT BASIC DISK&RETURN'. You can then insert the Master disk into drive 1 and press RETURN. Or you can use FORMAT1.COD (see Ch. 17) to format a disk with the BASIC on it and use this as your working disk. If your working disk has the BASIC and is in drive 1, just press RETURN. Either way, it takes about 16 seconds to reload the BASIC.

Entering long programs

When entering a long program you might get a NO ROOM error message. If that happens, you should save the program to the disk and then reload it using the following special format:

LOAD filename 1

The 1 after the filename instructs the system to keep the main part of the program on the disk. Then you can continue typing in your program. LIST, DEL, etc. will work as before; however, when you next save, you must use a different name. Also you will need to keep this disk in the computer during the save. If you have a one disk drive system, you must save back to the same disk. However, you can use COPYFILE.COD to move the program to another disk.

KILL, RENAME, LOCK, and UNLOCK

Descriptions of the above commands are in the reference manual. Each one can be called directly from the BASIC without affecting the program in memory.

SOUND

Consider the following format:

SOUND VOICE%,FREQUENCY%,DISTORTION%,VOLUME%

ATARI computers have four independent sound channels called voices; they are numbered 0, 1, 2, and 3. The value of VOICE% determines which voice you are issuing a command to. The value of FREQUENCY% (1 to 255) determines the sound frequency. For example, 121% is middle C. (See reference manual for a complete table). The value of DISTORTION% ranges from 0% to 15%. It controls the amount and type of noise output; 10% produces a pure note. The value of VOLUME% also ranges from 0% (sound off) to 15% (highest volume). Once started, a channel will continue to emit the same sound until another command is given to that channel, or until the program ends.

ASOUND and SCONTROL

You can use ASOUND and SCONTROL to set up a series of notes and then allow the computer to play the series automatically without any further commands. For example, the program starts a tune and then does something else, like manipulate a player, while the system is playing the specified tune. The formats for these commands are:

ASOUND VOICE%,ADR(linenumber)

SCONTROL integerexpress,integerexpress,integerexpress,integerexpress

In ASOUND the value of VOICE% specifies which voice you are issuing a command to (0, 1, 2, or 3). The number after ADR is the linenumber where the data for the voice is located. The data may extend over several successive lines and specifies frequency, duration, distortion, and volume. ASOUND does not start a voice; it only specifies what is to be played.

SCONTROL starts and stops the voices. The four numbers following SCONTROL correspond to the four ATARI voices. If the number is one, the voice is started or continued. If it is zero, the voice is stopped or not started. Thus you use ASOUND commands to specify the data for the voices and SCONTROL to simultaneously start them. This allows you to synchronize the voices. Consider the following program, which plays two notes and stops:

```
10   ASOUND 0%,ADR(1000)
20   SCONTROL 1%,0%,0%,0%
30   WAIT 60%
40   GOTO 5000
1000 CODE"5,!121,!168,7,!243,!170,0,FF"
5000 END
```

The 0% following ASOUND specifies that the sound will be produced on voice 0. Line 1000 is where the data for the voice is located. The CODE command at line 1000 is a special way of entering data into a program. It is used mainly with assembly language code. Here we are using it to enter a series of numbers separated by commas. If preceded by ! the numbers are in

decimal; otherwise, they are in hexadecimal. Fear not, however; you don't need to understand hex numbers to enter data. You do need to know that any number from 0 to 9 is the same in hex and decimal; thus, you don't need to use ! in front of them.

The data most commonly used with ASOUND is a three number group; look at the first group on line 1000 above. The first number must be from 1 to 254; it specifies duration in sixtieths of a second. It's a 5, meaning 5/60 of a second. The second number specifies frequency (1 to 255). It is 121, or middle C. (See reference manual for complete table.) The third number is a combination of two values (each ranging from 0 to 15) representing distortion and volume. To get the third number, multiply distortion by 16 and add the volume. Remember that a distortion of 10 produces a pure note. 168 equals 10*16+8, and thus gives a pure tone at about half volume. Since 15 produces maximum volume, 8 gives about half volume.

Now look at the second group of three numbers on line 1000. The 7 means that this sound will play for 7/60 of a second. The 243 means it is a low C. The 170 (10*16+10) means it is a pure tone of slightly over half volume.

Notice that the last group on line 1000 starts with a zero. Since it does not make any sense to play a note for 0/60 second, this zero is a signal that what follows is a special command. There are three of these: 0,FF tells the system to stop the sound on that channel, 0,FE stops all channels, and 0,FD is explained later in this chapter. Back to line 1000 again; 0,FF turns off the sound after the 7/60 second low C.

If you need to know where you are in the sound sequence, give an RTIME at the start, and use the TIME function to determine how many sixtieths of a second have elapsed.

The SCONTROL on line 20 starts the sound. The WAIT 60% on line 30 prevents the program from reaching the END before the tune has been played. Note that all voices are turned off by the END command.

It is easy to play a "tune" just once, as in the previous program. But how do you play a tune a limited number of times or repeat it continuously? And how do you put attack and decay into notes? You can use the tools described above, but it will be rather painful. Advan BASIC solves these problems with another special command. Consider the following example:

```
1000 CODE"FF,#2000,FF,#2000,FF,#2000,0,FF"
2000 CODE"5,!121,!168,7,!170,0,FD"
```

Line 1000 has three special commands which act somewhat like a GOSUB. Each command starts with FF, followed by a comma, a # symbol, and a linenumber. # means that the next number is a linenumber. FF is the hexadecimal form of 255; we use it because it's shorter. When the system detects a group starting with FF, it switches to the line specified by the number after # (in this case, line 2000). The data may lie on just one line or extend over several successive lines. 0,FD (at the end of line 2000) acts like a RETURN and causes the system to return to line 1000 and continue on that line, where it will again be sent to line 2000. Thus, the three special commands in line 1000 cause the sound pattern in line 2000 to be played three times. Here is another example:

```
10   ASOUND 0%,ADR(1000)
20   SCONTROL 1%,0%,0%,0%
30   WAIT 60%
40   GOTO 5000
1000 CODE"FF,#2000,FF,#3000,FF,#2000,0,FF"
2000 CODE"1,!121,!164,2,!121,!168,1,!121,!164,0,FD"
3000 CODE"1,!243,!164,2,!243,!168,1,!243,!164,0,FD"
5000 END
```

ASOUND and SCONTROL cause the data in the CODE statements to be used. On line 1000, the first FF switches control to line 2000--a middle C with varying volume. The 0,FD in line 2000 returns control to line 1000, where the next FF switches control to line 3000--a low C of varying volume. The 0,FD in line 3000 returns the system to line 1000, where the third FF switches control to line 2000 for the second time. The final 0,FF turns off voice 0. The WAIT command prevents the program from ending before the sound ends. One more example:

```
100  ASOUND 0%,ADR(1000)
110  SCONTROL 1%,0%,0%,0%
120  GOTO 120
1000 CODE"5,!121,!168,7,!243,!170,FF,#1000"
```

This is the same type of program as the first example in this chapter. In that example, we ended with 0,FF which stopped the channel. Here we end with FF,#1000. This will send the system back to the start of line 1000 and cause the sound to be repeated continuously. To stop the sound, you will need to insert into the program SCONTROL and 0% for that channel. To stop the program press BREAK.

Warning: The CODE command is a very powerful and efficient way to provide data for ASOUND, but it is also very dangerous. You can crash the system if you allow the program to try to execute CODE lines. Before the CODE statement, use END or GOTO, to go around it.

13. GRAPHICS

Graphics Modes

If you use the optional screen design and fine scrolling package, Advan BASIC will support most of the large number of ATARI graphics modes; this includes the 16 supported by the ATARI operating system, as well as many more. Without this optional package, Advan BASIC supports the same 16 graphics modes as ATARI; these are briefly described in Table 13-1.

Table 13-1

MODE	SIZE	TYPE	# OF COLORS
0	40x24	text	2 (1 color and 2 luminance values)
1	20x24	text*	5
2	20x12	text*	5
3	40x24	graphics	4
4	80x48	graphics	2
5	80x48	graphics	4
6	160x96	graphics	2
7	160x96	graphics	4
8	320x192	graphics	2 (1 color and 2 luminance values)
9	80x192	graphics	16 (1 color and 16 luminance values)
10	80x192	graphics	9
11	80x192	graphics	16
12	40x24	text**	4
13	40x12	text**	4
14	160x192	graphics	2
15	160x192	graphics	4

*With alternate character sets, these modes can provide good graphics
**Works like a text mode, but much more useful with alternate character sets in a graphics display.

The GRAPHICS command is used with a number which specifies the graphics mode. For example, the following line switches the display to graphics mode 4

```
10 GRAPHICS 4%
```

By adding special numbers to the desired mode, however, you can specify any of the following conditions:

A. An alternate character set (add 128 to mode number).

B. Player-missiles are used (add 64 to mode number).

C. The display is not to be cleared when it is opened (add 32 to mode number).

D. There is to be no text window at the bottom of the display (add 16 to mode number).

And you can combine several of the above options:

```
10 GRAPHICS 85%
```

Since 85=5+16+64, the display will be opened in mode 5 with no text window and with player-missiles activated.

Special note for users who do not have an XL or XE:
The system will remove the BASIC whenever you use the GRAPHICS command. This frees the memory needed for the graphics. When the program comes to an end, you will receive the message: INSERT BASIC DISK&RETURN. At this point, you can insert the Master disk into drive 1 and press RETURN. It takes about 16 seconds to reload. Or you can format a disk with the BASIC on it and use this as your working disk. In this case, you would not have to switch disks. To format a disk with BASIC use the utility FORMAT1.COD (See Ch. 17).

PLOT and COLOR

Both PLOT and PRINT are used to display information; PLOT is normally used for graphics data and PRINT for text data. If you are using a graphics mode with a text window, PRINT will put characters into the text window and PLOT will send data to the main display. The format for PLOT is

```
PLOT integerexpression,integerexpression
```

The first integerexpression specifies the column (horizontal position), and the second specifies the line (vertical position). Remember, column 0 is the left most column, and line 0 is the top line. The following line plots a point at the 6th column and the 3rd line:

```
10 PLOT 5%,2%
```

The color of the point is set by the COLOR command. Here is its format with an example:

```
COLOR integerexpression
```

```
10 COLOR 2%
```

In a text mode, the number following the COLOR command specifies the character to be displayed and, in some text modes, also gives some color information. In a graphics mode, the number following COLOR determines the color that the succeeding plot commands will place on the screen. Once the color has been set, it will remain until you give another COLOR command or a PRINT command. So to display a figure with only one color, you normally give a COLOR command and a series of PLOT commands.

SETCOLOR and PSETCOLOR

The actual colors which appear on the display are determined by what is in a set of registers called color registers (also called play fields). The ATARI operating system sets up these registers for certain colors when the mode is opened. In mode 4 for example, COLOR 1% gives an orange. There are five color registers for the main display and four for players and missiles. SETCOLOR changes the color that is produced by the main display color registers, while PSETCOLOR changes the player-missile registers. Here is the format and an example:

SETCOLOR integerexpression,integerexpression,integerexpression

10 SETCOLOR 1%,8%,4%

The first integerexpression after SETCOLOR specifies the main display color register and must be 0, 1, 2, 3, or 4. PSETCOLOR has the same format and works the same way, except that the first integerexpression refers to the player-missile color register and must be 0, 1, 2, or 3. The second integerexpression specifies the color, as shown in the following table:

Table 13-2

NUMBER	COLOR
0	gray
1	gold
2	orange
3	red orange
4	pink
5	purple
6	purple blue
7	cyan
8	blue
9	light blue
10	turquoise
11	blue green
12	green
13	yellow green
14	orange green
15	light orange

The third integerexpression gives the luminance (brightness) and must be an even number from 0 (darkest) to 14 (brightest). Note that luminance has a big effect on color. For example, zero hue is called gray, but zero hue with 0 luminance is black and zero hue with 14 luminance is white.

To illustrate how these commands work, consider mode 3, a 4 color graphics mode. In this mode the number following COLOR must be 0, 1, 2, or 3. If you use a COLOR 1% command, the color plotted is specified by color register 0 (note: not register 1), COLOR 2% displays what is specified by color register 1, COLOR 3% displays what is specified by color register 2, and COLOR 0% what is specified by color register 4. The border is also controlled by color register 4. Unless you use SETCOLOR to change them, the colors registers will have:

Color register	0	orange
	1	light green
	2	blue
	4	black

Now, if you want to shift from light green to purple for COLOR 2% (i.e., color register 1) you need to give the following command:

SETCOLOR 1%,5%,8%

Modes 5, 7, and 15 work just like mode 3 in the way the COLOR command works with color registers.

Modes 4, 6, and 14 are graphics 2 color modes. Thus, the number following COLOR must be 0 or 1. COLOR 1% refers to color register 0 and COLOR 0% refers to color register 4.

Graphics mode 8 is a one color mode with two luminances. COLOR 0% specifies the color and luminance from color register 2. COLOR 1% specifies the color from register 2 and the luminance from color register 1.

Mode 9 is a one color mode with 16 luminances. The color comes from color register 4. The luminance comes directly from the number in the color command. Thus, COLOR 15% will give the color from color register 4 at maximum brightness, while COLOR 0% gives the same color at minimum brightness.

Mode 11 is a one luminance mode with 16 colors. The luminance comes from the luminance of color register 4. The color comes directly from the number in the color command using the coding shown in Table 13-2. Thus, COLOR 8% will give BLUE with a luminance specified by color register 4.

Mode 10 is a 9 color mode. It uses the 4 player-missile registers as well as the 5 main display color registers. Table 13-3 shows how this works.

Table 13-3

Number in Color Command	Color Register
0	player-missile 0
1	" " 1
2	" " 2
3	" " 3
4	main display 0
5	" " 1
6	" " 2
7	" " 3
8	" " 4

Modes 1 and 2 are normally text modes. If you use an alternate character set, however, they can produce effective graphics displays. The characters displayed are double width (compared to mode 0) and can be in one of four colors. Mode 2 characters are double the height of mode 0 characters. You can display all characters with ASCII codes from 32 to 95. (See Appendix A). One way to work with these modes is to use the +16 option in the GRAPHICS command and then use the PRINT command. For example, the command PRINT "A" will display a capital A using the color specified by color register 0. PRINT "a" will also display a capital A; however, the color is determined by color register 1. The following chart shows how to get capital letters with different colors:

Letter entered	Color from color register
upper case	0
lower case	1
inverse upper case	2
inverse lower case	3

In all cases a capital letter will be displayed. Numbers and symbols are tougher since we don't have lower case for them. Numbers and symbols entered in inverse mode will use the color from color register 2, while those entered in the normal mode use color register 0.

If you need numbers and/or symbols with color registers 1 or 3, use the following chart:

ASCII Code	Color Register	Color to plot
32 to 63	1	ASCII code -32
32 to 63	3	ASCII code +96
64 to 95	1	ASCII code +32
64 to 95	3	ASCII code +160

For example, to display an @ (ASCII code=64).using color register 3, you could use:

```
PRINT CHR$(224%);  or   COLOR 224%
                        PLOT X%,Y%
```

Modes 12 and 13 are called character modes, but with an alternate character set they are effective graphics modes. The best way to use them is with the Advan optional screen design package, which lets you design a custom alternate character set.

DRAWTO and FILL

The DRAWTO command will draw a line from the last point displayed to the point specified. The format is

DRAWTO integerexpression,integerexpression

The first integerexpression gives the column and the second the linenumber of the point you are drawing to. In the following program, lines 10 and 15 set the graphics mode and color. Line 20 plots a point at 5,5. Lines 30-60 draw the sides of a square:

```
10 GRAPHICS 3%
15 COLOR 2%
20 PLOT 5%,5%
30 DRAWTO 10%,5%
40 DRAWTO 10%,10%
50 DRAWTO 5%,10%
60 DRAWTO 5%,5%
70 WAIT 160%
```

You can use the FILL command to fill up the square. It works like DRAWTO, except that as it plots a point, it also fills in all the points to its right until it runs into the screen edge or another plotted point. Its format is the same as DRAWTO. To fill in the above square, change line 60 to:

```
60 FILL 5%,5%
```

This causes the system to draw a line from 5,10 to 5,5 and to fill in the area to the right of the line.

DFILL

You can use DFILL to fill the entire screen, a player, or a missile with a given number. Its format is

DFILL integerexpression,integerexpression

The second integerexpression equals the number which is used to fill the screen, player, or missile. The first integerexpression determines what is filled. The following table shows how this works.

Table 13-4

Value of integerexpression	Object filled
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3
16	main display screen

Normally, filling with a zero will clear the screen, player, or missile. For example, the following line will clear the display screen:

```
100 DFILL 16%,0%
```

POS and LOCATE

The POS command is used to position the cursor. In mode 0 the cursor is visible; in the other modes it is not. This command is normally used before a PRINT command and determines the location of the characters printed. The LOCATE function lets you determine what is on the main display screen at a given point. See the reference manual for more information on these two commands.

14. PLAYER-MISSILES

The ATARI operating system sets all player-missile color registers to black, so until you give a PSETCOLOR command, your player-missiles will be invisible. PSETCOLOR and DFILL were described in the previous chapter. Remember that you must add 64 to the mode in the GRAPHICS command to activate player-missiles. You control the four players and four missiles by using a group of Advan BASIC special commands. Table 14-1 shows how the first integerexpression in each command determines the player-missile number.

Table 14-1

Integerexpression #	Player-missile #
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3

PSIZE

Players at normal width are twice as wide as a mode 0 character. They have 8 points which can be turned on or off. If on, they will have the color you set for the player. If off, they will be transparent; that is, the color will be that of the main display. Missiles at normal width are 1/2 the width of a mode 0 character, and they have 2 points which can be turned on or off. Their color is the same as that of the player with the same number (e.g., missile 2 and player 2 have the same color). Use the PSIZE command to change a player-missile's size. Note that this changes the width of each point, but not the number of points. The format is

PSIZE integerexpression,integerexpression

The first integerexpression determines which player or missile size is being set. (Table 14-1). The second integerexpression determines the size according to the following chart:

Integerexpression Value	Player-missile Size
0	normal size
1	doubled
3	quadrupled

For example, the following line sets missile 2 to quadruple size:

```
10 PSIZE 6%,3%
```

HPOS

You can set the horizontal position of a player with HPOS, which has the

following format:

HPOS integerexpression,integerexpression

The first integerexpression determines the player or missile number (Table 14-1). The second integerexpression sets the horizontal position of the player or missile. The left and right boundaries of most displays are approximately 40 and 216.

PDISPLAY

Use the PDISPLAY command to place a figure in a player or missile. The format is

PDISPLAY integerexpression,ADR(linenumber),integerexpression

The first integerexpression specifies the player-missile number. The second integerexpression specifies the vertical location of the top of the figure. 128 is the center of the screen, zero is the top, and 255 the bottom. The linenumber is where the data defining the figure is located.

Suppose you are defining a figure for a player. Remember that players are 8 display points wide and they may be from 1 to 253 vertical lines long. Actually, on an average TV monitor, the top and bottom will be cut off. The first piece of data must be the number of vertical lines your figure will occupy. Then you need to give data for each line, starting with the top line. You have to specify which points are on and which are off. To do this for each line, you can use a & symbol followed by 8 characters, each a 0 or 1. (Since missiles are only 2 display points wide, they require only 2 characters after the & symbol). Each '1' means that the corresponding point in the player is on (its color is that of the player's color register). Each '0' means that the point is transparent. The CODE command is used to enter the data. The following line is an example of how this works; it has the data for a box, 4 lines high.

```
CODE"4,&11111111,&10000001,&10000001,&11111111"
```

The '4' tells how many lines and is followed by the data for each line. Note that commas are used to separate data items. The first and fourth data items have all ones, thus representing solid horizontal lines. The second and third data items have the end points on and the other points off, thus forming part of the left and right edges. Here is the data statement used in a complete program:

```
10  PSETCOLOR 1%,3%,8%
20  PSIZE 1%,0%
30  GRAPHICS 67%
35  DFILL 1%,0%
40  PDISPLAY 1%,ADR(100),128%
50  HPOS 1%,128%
60  WAIT 160%: GOTO 200
100 CODE"4,&11111111,&10000001,&10000001,&11111111"
200 END
```

This program sets player 1 to red orange at normal size. Line 30 sets GRAPHICS mode 3+64 which activates the player-missiles. Line 35 clears the player and then line 40 puts the data on line 100 into the player. The

top of the figure is set at approximately mid screen (128). Line 50 sets the player horizontal position to 128 (about mid screen). The WAIT 160% gives you time to look at it. Note that you must go around the CODE command. If the BASIC runs into the CODE command, the system will probably crash, and you will have to reload it. Note that the system waits until the vertical blank interrupt to insert the data into a player or missile.

You might recognize the fact that the & symbol tells the system that the following data is in binary. If you know hex, you can convert binary to hex and save some typing, plus squeeze more into a line. However, compiled code length is the same. Note that in a CODE statement, hex numbers do not need to be preceded by any special symbol. The following is how line 100 looks with hex numbers:

```
100 CODE"4,FF,81,81,FF"
```

You can move a player or missile with a series of HPOS or PDISPLAY commands. Also the system has a built-in mechanism for automatically moving a player or missile, and even for automatically changing the figure.

I should mention that PDISPLAY places the figure into the player or missile at the specified vertical location. It does not erase the rest of the player or missile. So, if you are moving a player vertically, you need to erase the part of the figure which is not overwritten. One way to do this is to put extra blank lines on the top and bottom of the figure. For example, you can define the box in the previous program with the following line. The length of the player is changed from 4 to 8. Note that you don't need to use a & symbol for a blank line; a zero will do:

```
100 CODE"8,0,0,&11111111,&10000001,&10000001,&11111111,0,0"
```

The following example will move the box across the screen from the upper left to the lower right. Note that each time the figure is put into player 1, it will erase the previous figure:

```
10  PSETCOLOR 1%,3%,8%
20  PSIZE 1%,0%
30  GRAPHICS 67%
35  DFILL 1%,0%
40  FOR T%=0% TO 240% STEP 2%
50     PDISPLAY 1%,ADR(100),T%
60     HPOS 1%,T%
70  NEXT T%
80  END
100 CODE"8,0,0,FF,81,81,FF,0,0"
```

DFILL

DFILL (described in Ch.13) was used in line 35, above, to clear the player. Filling a player-missile with 1% turns on only the first point on the right. Filling with 2% turns on the second point, 4% the third point, 8% the fourth point, etc. You can add together the numbers shown above to turn on groups of data points. For example, DFILL 1%,3% will turn on the first two data points on the right and turn off the others for player 1. The net effect is a vertical bar one quarter the width of the player.

Automatic Horizontal and Vertical Player-Missile Movement

PRATE

To use the built-in mechanism for automatically moving a figure, you need to specify its horizontal and vertical speed using the PRATE command. Its format is

PRATE integerexpress,integerexpress,integerexpress,integerexpress

The first integerexpression determines which player-missile you are working with (Table 14-1). The second integerexpression sets the horizontal speed and the third integerexpression sets the vertical speed. The fourth integerexpression sets the rate at which changes are made in the figure itself. I will discuss this later in the chapter. For now, let's just keep it 0. Speeds around 256 provide moderate and smooth motion. 32767 is the maximum. Motion is so fast for speeds greater than a few thousand, that the effect is rather weird. Speeds which divide evenly into 256 (e.g., 128, 64, etc.) and speeds which are multiples of 256 (e.g., 512, 768, etc.) give the smoothest motion. Positive horizontal speeds give motion to the right and negative give motion to the left. Positive vertical speeds give downward motion and negative give upward motion. Note that figures going off one edge of the screen will reappear at the opposite edge.

Sometimes you want motion to begin immediately after the PRATE command and sometimes you want to wait. For example, you may be trying to synchronize the motion of several figures. If you want the PRATE command to start the motion, you must add 256 to the first integerexpression. The following line sets movement rates for player 1 and then actually starts the motion:

```
100 PRATE 257%,256%,256%,0%
```

You can also use PRATE to stop the automatic motion of a player or missile by adding 512 to the first integerexpression. The following line stops the automatic motion of player 1:

```
100 PRATE 513%,0%,0%,0%,
```

I should mention that you can use PRATE to change the speed of a player already moving. For example, the following line changes the horizontal speed of player 1 to 512:

```
100 PRATE 1%,512%,256%,0%
```

PCONTROL

PCONTROL lets you synchronize the movement of players and missiles. You must first give PDISPLAY and PRATE commands for each player and missile you want to move. (In the PRATE commands, do not add 256.) Then you can use the PCONTROL command to simultaneously start or stop all players and missiles. The format of PCONTROL is

PCONTROL integerexpress,integerexpress,integerexpress,integerexpress

The first integerexpression controls player-missile 0, the second controls player-missile 1, etc. The following chart shows how the value of the

integerexpression determines the motion.

Value of Integerexpression	Effect
0	both player and missile cease motion
1	player starts and missile stops
2	missile starts and player stops
3	both player and missile start

Here is the box program again; as before, the box will be moving down and to the right. To vary the speed, change the numbers in the PRATE command. To end the program press the BREAK key.

```
10  PSETCOLOR 1%,3%,8%
20  GRAPHICS 67%
30  DFILL 1%,0%
40  PDISPLAY 1%,ADR(100),128%
50  HPOS 1%,128%
60  PRATE 1%,256%,256%,0%,
70  PCONTROL 0%,1%,0%,0%
80  GOTO 80
100 CODE"8,0,0,&11111111,&10000001,&10000001,&11111111,0,0"
```

COLL

If you are firing bullets or rockets across the screen, automatic missile movement is very handy. Often, however, you want to know if the missile has hit anything. The COLL (collision) command provides this information. See the reference manual for a description of COLL.

Locating players and missiles

If you are automatically moving a player or missile, the following table gives the memory locations for their horizontal and vertical positions. For example, PEEK(1134) gives the horizontal location of missile 2.

Table 14-2

		memory location horiz. position	memory location vert. position
player	0	1128	1152
	1	1129	1153
	2	1130	1154
	3	1131	1155
missile	0	1132	1156
	1	1133	1157
	2	1134	1158
	3	1135	1159

Automatic modification of a player or missile figure

Suppose you want to use a player to display a moving stick. If the stick is rotating end over end, you need to change the figure as well as have it move horizontally and vertically. The same is true of a person running. In the case of the stick, you could show it in a horizontal position for 2/60

second, at a 45 degree angle for 2/60 sec., vertical for 2/60 sec., at a 135 degree angle for 2/60 sec., and then repeat the sequence. You can do this with the PRATE command and appropriate data. The last integerexpression in the PRATE command tells how long in sixtieths of a sec. to keep each figure before changing the display (maximum is 255). However, zero is special and means no change. Thus, if you set the last integerexpression to 2%, each figure will remain on the screen for 2/60 sec.

Of course, now you'll need data for each of the figures. The way you do this is to put the data for one figure right after the data for the previous figure. For the stick problem, you'll need four sets of data. Each set follows the previous set, and each starts with the vertical length. They can be on one or several lines. Each line must start with a CODE command. Suppose you plan to use the following four figures for the stick. The one's represent the turned on part of the player:

00000000	00000000	00011000	00000000
00000000	01100000	00011000	00000110
00000000	00110000	00011000	00001100
11111111	00011000	00011000	00011000
11111111	00001100	00011000	00110000
00000000	00000110	00011000	01100000
00000000	00000000	00011000	00000000

Here is the rotating stick program. Note that you don't need to use leading zeroes with the & symbol:

```
10  PSETCOLOR 1%,3%,6%
20  GRAPHICS 67%
30  DFILL 1%,0%
40  PDISPLAY 1%,ADR(100),128%
50  HPOS 1%,128%
60  PRATE 1%,256%,256%,2%
70  PCONTROL 0%,1%,0%,0%
80  GOTO 80
100 CODE"!12,0,0,0,0,0,&11111111,&11111111,0,0,0,0,0"
110 CODE"!12,0,0,0,&1100000,&110000,&11000,&11000,&1100,&110,0,0
,0"
120 CODE"!12,0,0,&11000,&11000,&11000,&11000,&11000,&11000,
&11000,&11000,0,0"
130 CODE"!12,0,0,0,&110,&1100,&11000,&11000,&110000,&1100000,0
,0,0"
140 CODE"FF,#100"
```

This is the same as the previous program, except for the data and the last integerexpression in PRATE. First, the system displays the data from line 100 and after 2/60 second, the figure defined on line 110. After another 2/60 second, it displays the figure from line 120 and then the figure from line 130. After displaying the line 130 data for 2/60 second, the system will go to line 140 for the next figure. Line 140 begins with a special code, FF (=255 in decimal). Because in Advan BASIC the vertical length of player-missiles may not exceed 253, the system interprets the FF as a special command and will look at the data right after it as a linenumber. The linenumber must have a # symbol in front of it. The system will switch immediately to the specified linenumber and continue on from there. Thus, the system displays the data from lines 100, 110, 120, and 130 as a repeating series.

Note that there is a ! symbol in front of the 12. This tells the system that the following number is in decimal, just as a & symbol indicates binary. Why didn't we use a ! symbol in front of the 8 in the previous example? Because the system expects hex numbers in a CODE command, and the numbers 0 through 9 are the same in hex and decimal.

Suppose you want to display lines 100, 110, 120, 130, 120, 110, 100 as a repeating series. You can do this by making a new line, say 134, the same as line 120, and making 138 the same as 110, and so forth. A more convenient way is to use another special command, FE (=254 in decimal). Again, because in Advan BASIC the vertical length of player-missiles may not exceed 253, the FE command is a signal; what follows must be a comma, # symbol, and linenumber. At the FE command, the system goes to the specified linenumber only for one figure and then returns to the data right after the FE command for the next figure's data. So to display the lines as a repeating series, change lines 40 and 140 in the previous program to:

```
40 PDISPLAY 1%,ADR(140),128%

140 CODE"FE,#100,FE,#110,FE,#120,FE,#130,FE,#120,FE,#110,FF,#140"
```

Finally, suppose you want to display this set of lines only once; that is display 100, 110, 120, 130, 120, 110, 100 and then stop. You can do this by using the special command, zero:

```
140 CODE"FE,#100,FE,#110,FE,#120,FE,"130,FE,#120,FE,#110,FE,#100,0"
```

If you do use this special command, the following table lets you determine if a player or missile has been stopped.

Table 14-3

player	0	PEEK(1075%)	and	1%
	1	PEEK(1075%)	and	4%
	2	PEEK(1075%)	and	16%
	3	PEEK(1075%)	and	64%
missile	0	PEEK(1075%)	and	2%
	1	PEEK(1075%)	and	8%
	2	PEEK(1075%)	and	32%
	3	PEEK(1075%)	and	128%

If the expression in the right hand column is zero, the player or missile is not moving. For instance, if PEEK(1075%) and 32% is zero, then missile 2 is at rest; otherwise it is moving.

15. DISPLAY LIST INTERRUPTS

ATARI computers have a nice feature called display list interrupts. I know of no other moderately priced personal computer that has anything like it. Using these interrupts, you can modify the display at the start of one or more screen lines. For example, at a given line you can change the value of a color register or the horizontal position of a player or switch to an alternate character set. Something this good shouldn't go to waste, and so Advan BASIC has two special commands designed to take advantage of these display list interrupts.

SETINT@

The format of a SETINT@ command is

SETINT@ integerexpress,integerexpress,integerexpress,integerexpress

The first integerexpression gives an identifying number (0 to 7) to the interrupt, so that you can refer to it. The second integerexpression specifies the display list line on which the interrupt is to occur. The third integerexpression specifies the location of what you would like to change and the fourth integerexpression gives the new value for this location. Table 15-1 lists some locations you might want to change at an interrupt.

TABLE 15-1

Memory Location	Function of Location	
53266	color register of player missile	0*
53267		1
53268		2
53269		3
53270	color register (play field)	0
53271		1
53272		2
53273		3
53274	color register (background)	4
53248	horizontal position of player	0
53249		1
53250		2
53251		3
53252	horizontal position of missile	0
53253		1
53254		2
53255		3
54281	location of alternate character set	**

*color number times 16+luminance is stored here
** must be a multiple of 1024; i.e., 1024*30=30720. The value stored at 54281 should be the alternate character set address divided by 256. See Appendix D for a memory map and possible locations for an alternate character set.

I should mention that you can only put one interrupt at a given line. The following program draws and fills in a square. It uses mode 4 which has only two colors. But by changing color register 0 at two interrupts, you can have a tri-colored square. Actually, you can put 8 colors plus the background on the screen.

```
10 GRAPHICS 4%:COLOR 1%
20 PLOT 10%,5%
30 DRAWTO 60%,3%
40 DRAWTO 60%,45%
50 DRAWTO 10%,45%
60 FILL 10%,5%
70 SETINT@ 0%,16%,53270%,7%*16%+8%
80 SETINT@ 1%,32%,53270%,12%*16%+8%
90 GOTO 90
```

Lines 10 through 60 set the graphics mode and draw a rectangle. Lines 70 and 80 set two display list interrupts. For all of the graphic modes, the first three display list lines (0, 1, and 2) are blanks. That is, the 16% at line 70 sets a display list interrupt at display list line 16, which is screen line 13. At that line, color register 0 is switched from red to blue. At screen line 29, color register 0 is again changed; this time to green. Note that the change does not occur until the end of the line on which the interrupt is set. This is to avoid making a change during a line. The change from red to blue occurs at the start of line 14, and the switch to green at the start of line 30.

You might wonder why you didn't need to reset the color back to reddish-orange. The reason is that during a vertical blank (at the end of a display frame), all the color registers, as well as the alternate character set locations, are reset from special memory locations.

Before trying the above example, you will need to append the display list interrupt subroutines. Like PRINT USING, there isn't enough room to fit in these special routines. To use them, insert a Master disk (or another disk with the program) and then type APPEND DLISTINT.APP.

To remove an interrupt, set the last three integerexpressions to zero. The following command removes interrupt 1:

```
SETINT@ 1%,0%,0%,0%
```

CINT@

Format:

```
CINT@ integerexpression,integerexpression
```

The first integerexpression is the identifying number of the interrupt. CINT@ is used to change the value which is stored when the interrupt occurs. You can use CINT@ only after SETINT@ has been used. The second integerexpression gives the new value to be stored by the interrupt. You could use SETINT@ itself to change this value; however, SETINT@ has to wait until just the right point in the display to put the interrupt into place. CINT@ is much faster because it can make an immediate change.

```
10  GRAPHICS 68%
20  PSIZE 2%,0%
30  DFILL 2%,0%
40  PDISPLAY 2%,ADR(200),60%
50  PDISPLAY 2%,ADR(200),120%
60  PDISPLAY 2%,ADR(200),180%
70  SETINT@ 0%,3%,53268%,3%*16%+8%
80  SETINT@ 1%,4%,53250%,0%
90  SETINT@ 2%,10%,53268%,7%*16%+8%
100 SETINT@ 3%,17%,53250%,0%
110 SETINT@ 4%,31%,53268%,12%*16%+8%
120 SETINT@ 5%,32%,53250%,0%
130 FOR T%=0% TO 255%
140     CINT@ 1%,T%
150     CINT@ 3%,255%-T%
160     CINT@ 5%,T%+128%
170 NEXT T%
180 GOTO 130
200 CODE"4,&100100,&100100,FF,FF"
```

The above program puts the figure defined in line 200 into player 2 at three different vertical sections. Six display list interrupts are set. Three are used to change the player color and three to change the horizontal position. The FOR loop modifies the horizontal player position and causes two sections of the player to move right and one to move left.

Special Note: Because of timing problems, display list interrupts in modes 8, 9, 10, 11, 14, and 15 cannot be in adjacent display list lines. There must be at least one display list line between interrupts in these modes.

16. MACHINE LANGUAGE SUBROUTINES

Advan BASIC allows you to insert assembly language codes easily into a BASIC program. You can even use mnemonics for the various 6502 commands. Another nice feature is that you can access program variables by name in your assembly language code. If you don't know how to program in assembly language, much of the following information will be difficult to understand. Probably you should pick up a book on 6502 assembly language programming before attempting to write an assembly language program.

Since Advan BASIC is compiled, most programs written in it will run considerably faster than in a non-compiled BASIC. There are situations, however, where you want maximum possible speed. Then you need the Advan BASIC machine language subroutine capability.

MACHINE

The MACHINE command tells the compiler that the information which follows will be machine or assembly language code. Its format is

MACHINE linenumber

When the machine language program comes to an end, normally with an RTS command, control transfers back to the BASIC code. Execution resumes at the linenumber specified in the MACHINE command.

CODE

The CODE command is used in several ways in the BASIC. It provides a way to enter data for the sound routines and for the PDISPLAY command. Its primary purpose, however, is to allow the entry of assembly language code. For example, LDA is the standard mnemonic code to load a number from a given source into the accumulator. Appendix E lists the 6502 mnemonics used by the Advan compiler.

```
CODE"LDA,FF,9F"
```

In the above line, the compiler translates LDA into machine code. FF and 9F are hex numbers which give the address of the number loaded into the accumulator. Note that, as is standard in 6502 code, the FF is the least significant part of the address, while 9F is the most significant part (i.e., the address is 9FFF). The following program will store an S in the lower right hand corner of the display screen:

```
100 MACHINE 200
110 CODE"LDAIM,33,STA,FF,9F,RTS"
200 END
```

The LDAIM assembly language code causes the 6502 to load the next number (33) into the accumulator. The STA causes the 6502 to store the accumulator (33) into memory location 9FFF, which corresponds to the lower right hand corner of the screen. In an ATARI computer, the hex number 33 is the screen code for an S. The RTS signals the end of the assembly language subroutine and causes control to transfer to line 200, as specified by the 200 in the MACHINE command. The following segment

illustrates another feature of the CODE command:

```
100 MACHINE 200
110 CODE"LDA,FF,9F,CMPIM,34,BEQ,@120,LDAIM,34,STA,FF,9F"
120 CODE"RTS"
200 END
```

At line 110 the number at memory location 9FFF is loaded into the accumulator. The CMPIM is the assembly language mnemonic for comparing the accumulator with the following number (34). BEQ is the code for "branch if equal". @ followed by a linenumber gives the location to go to if the accumulator equals 34. The program will check what is in location 9FFF. If it is 34, it will branch to line 120 and execute the RTS. If it is not, it will load 34 into the accumulator, store the 34 into location 9FFF, and execute the RTS. Whenever you use a branch command (e.g., BNE,BCC), it should be followed by a comma and then @ followed by a linenumber. Here is another example:

```
100 MACHINE 200
110 CODE"LDA,FF,9F,CMPIM,34,BEQ,@120,LDAIM,34,JMP,#130"
120 CODE"LDAIM,33"
130 CODE"STA,FF,9F,RTS"
200 END
```

The above program is similar to the previous one. First it compares the number in 9FFF with 34. If they are equal, it goes to line 120 and loads 33 to the accumulator, executes the code on line 130 which stores the 33 to 9FFF, and finally returns. If they are not equal, it loads 34 to the accumulator and executes the JMP command. This is the assembler mnemonic for JUMP, and the next two bytes must specify where it is to jump. Normally you follow JMP, JSR, and JMPI with comma, # sign, linenumber. The compiler inserts two bytes which represent the address of the first command on the given line. Thus, after loading 34, it jumps to the STA,FF,9F on line 130, which stores the 34 to 9FFF, and executes the RTS command.

Warning: The value of the X register must not be changed by the routine. If you must use the X register, it must be saved and then restored before the final RTS. TXA,PHA saves the X register and PLA,TAX restores it.

CODEL

The CODEL command is normally used to specify the address of a variable. When used in this way, there are several possible formats:

```
CODEL(variablename)

CODEL(variablename+integernumber)

CODEL(variablename+"L")

CODEL(variablename+"H")
```

Integernumber is a positive integer without a % sign at the end. You can also use CODEL to specify the address of a line number. (See the reference manual for information on this option.) The following program inputs a number, adds one to it, and prints the result:

```
100 INPUT T%
110 MACHINE 200
120 CODE"INC":CODEL(T%):CODE"BNE,@130,INC":CODEL(T%+1)
130 CODE"RTS"
200 PRINT T%
```

Let's see how it works. On line 120, INC adds one to the number at the memory location specified by the following two bytes. CODEL generates the two byte address of the lower order 8 bits of T%. Thus, the lower order part of T% is increased by one. If the result is non zero, we are done and BNE causes a branch to line 130. If INC causes a zero, we have a carry and must add one to the high order part of T%. The CODEL(T%+1) generates the two byte code for one plus the address of T% (for the high order part of T%). In another example, the program stores zero in all elements of the array A%:

```
100 DIM A%(5)
110 T%=ADR(A%(0%))
120 MACHINE 200
130 CODE"LDYIM,!11,LDA":CODEL(T%):CODE"STAZ,E0,LDA"
140 CODEL(T%+1):CODE"STAZ,E1,LDAIM,0"
150 CODE"STAIY,E0,DEY,BPL,@150,RTS"
200 END
```

At line 110 we set T% equal to the address of A%(0%), which is the first element of the array. There are six elements in the array and therefore, 12 locations that we must zero. Remember, each integer takes up two bytes of memory. At line 120 we first load the Y register with 11. Note the ! symbol in front of the 11. Normally, the CODE command expects assembly language mnemonics or hex numbers. Any number preceded by !, however, is assumed to be a decimal number.

Next we load the low order part of T% (the low order part of the address of A% (0%)), and then store it in the zero page location E0. Note that any mnemonic ending in a 'Z' refers to a zero page location and needs only one hex byte following it to specify the location. Next we get the high order byte of T% and store this in zero page location E1. Finally on line 140, we load the accumulator with 0.

At line 150, STAIY is a store indirect command. The E0 immediately following it tells us the address where we are to store the accumulator; the address is the value in the Y register plus the number stored in E0 (low order part) and E1. In line 140 we stored T%, and thus the address of A%(0%) in E0 and E1. The Y register starts at 11 and therefore, we will store zero in the address of A%(0%)+11. The loop at 150 stores zero in all 12 bytes of the array A%. Note that an assembly language subroutine may use only the zero page locations from D4 to FF.

WARNING: If you make a mistake in assembly language code, the system might hangup. Even BREAK may not get you out of it. Therefore, be sure to save your program before running it.

17. UTILITY PROGRAMS

There are several helpful utility programs available on the Advan BASIC disks. To use them, first insert the Master disk into drive 1 and then type EXEC followed by a space and then the program name. Remember that any program you have in memory will be erased when you execute one of these utility programs. So if you don't want to lose a program in memory, save it.

CLEAN.COD

As you enter a program, Advan BASIC converts it to token form. For example, each command and most variables are assigned single byte codes, reducing the amount of room needed to save a program. In addition, a list is created of the variables entered. If a program is modified, a given variable may no longer be needed, but the variable is still in the variable tables. Also, other unused variables might be created if errors are made in entering lines.

As long as the maximum number of variables (255) is not exceeded, the unused variables will not cause problems and will not affect the length of the compiled program. They will, however, take up extra space on the disk and in the computer during a compile, although the difference is usually not great. Executing CLEAN.COD will produce a new file with the unused variables removed.

STATPROG.COD

This program finds the program length (not including data length), the number of program lines, and the number of variables used. It will also tell you how many unused variables are in the variable table (see CLEAN.COD for information on unused variables). One quite useful function of STATPROG.COD is to check if a variable has been used only once. This can happen if you misspell a variable name. Sometimes these errors are hard to spot; if STATPROG.COD finds one, it will give you the variable name and line number.

CHECKSUM.COD

This program is helpful if you want to send a program listing to a friend or a magazine. Also it provides a valuable hard copy (on paper) backup for your important programs. CHECKSUM.COD allows you to print out a copy of a program with a checksum for each line. The checksum appears as a ! symbol followed by four characters. It also prints out the number of lines and a checksum for the whole program.

To produce a program listing with checksum data, insert the Master disk and type EXEC CHECKSUM.COD; then choose option 1. Note that the WIDTH system command can be used to set the printer width.

If someone wants to enter your program from the listing with the checksum data, he types it in and saves it to the disk. Next, he types EXEC CHECKSUM.COD and chooses option 2, which checks a program. After he inserts the program disk and gives the program name, CHECKSUM.COD will provide a count of the number of program lines. If this does not agree

with the number on the listing, CHECKSUM.COD can display a list of the line numbers. This can be checked against the program listing.

If the number of lines is correct, CHECKSUM.COD will calculate a checksum for the total program. If this agrees with the total program checksum on the listing, he has probably entered the program correctly. Of course, no error check is perfect. The calculated checksum will catch things like reversed letters, but there are errors it will miss. If the checksum is correct, it's probably worth trying to run the program.

If the total program checksum is not correct, he can check one line at a time. When he finds a mistake, he will be asked to enter the correct checksum for that line. He will then be told if correcting this line brings the total program checksum into agreement with the listing checksum. Either way, he can continue checking lines.

COPYDISK.COD

This program allows you to duplicate a disk. If you have a one disk system, the program will tell you when to shift disks. Note that for one disk systems, you may have to switch disks several times before a disk is completely copied.

COPYFILE.COD

This program allows you to copy one or more files from one disk to another. If you have a one disk system, the program will tell you when to switch disks. When the program asks for the names of the files to be copied, you can specify a given file or use a wildcard option to specify several files.

There are two wildcards which you can use to substitute for the symbols in a file name; they are * and ?. The ? in a file name means that, when searching for matching files, the program will accept any symbol in the ? character position. For example, if you specify filename D2:ALPHA.??, files on disk 2 with names like ALPHA.11 or ALPHA.A7 will be copied. The * symbol is equivalent to a series of question marks filling in that section of the filename. For example, D1:ALPHA.* is the same as D1:ALPHA.??? and will copy all files on drive 1 starting with ALPHA. Also *.COD is the same as D1:????????.COD and will copy all files on drive 1 ending in .COD. If you are using wildcards in the input filename, you should use wildcards in the output filename, or you may get several files with the same name.

Special note: One way you can use this program is to copy all of these utility programs to another disk. Then you won't need to use the Master disk when you want to run a utility program.

FORMAT.COD

This program allows you to format single density disks.

FORMAT1.COD

This program allows you to format a disk in single density or in 1050 (so called 1 1/2 density) mode. In 1050 mode, the disk operating system allows you to use 940 of the possible 1040 sectors. FORMAT1.COD also gives you the option of putting a copy of the BASIC on the disk. This is useful mainly if you do not have an XL or XE computer. In that case, you will need

to remove the BASIC before running a long program or one with graphics, and then reload the BASIC after the program is executed. The system will ask you to insert a disk with the BASIC into drive 1 and then press RETURN. If you already have the BASIC on your working disk, however, you will not have to switch disks; just press RETURN and after about 16 seconds the BASIC will be reloaded. This will also reduce wear and tear on your Master disk.

RAMDISK.COD

This program allows 130XE owners to use the extra 64K of RAM as a RAM disk. After executing RAMDISK.COD, all references to disk four will access the RAM disk. In most cases, you can use this extra memory just like an ordinary disk. For instance, DIR D4: will give you a directory of the RAM disk without effecting any program in memory. Do not forget, however, that when you turn off the computer, or if the system locks up, you will lose anything in the RAM disk. Be especially careful with untested machine language code.

Note that RAMDISK.COD will automatically copy COPYFILE.COD from drive one to the RAM disk, if COPYFILE.COD is on the drive one disk. If you need more room in the RAM disk, you can delete this program using the KILL command.

SIEVE.BAS

This program demonstrates the speed of Advan BASIC. Type RUN SIEVE.BAS.

List of System Commands

APPEND	LLIST	RENAME
COMPILE	LMARGIN	RUN
DEL	LOAD	SAVE
DIR	LOADS	SAVEC
EXEC	LOCK	SAVES
KILL	NEW	UNLOCK
LIST	PEEK	WIDTH
	POKE	

NOTE: Filename is frequently used in the description of the system commands. Here are some examples of legal filenames.

ALPHA ALPHA.1C2 D2:BETA D3:BETA.COD D4:ABCDEFGH.111

If the file is not on disk one, the filename must start with D followed by the disk number (1 to 4) and then a colon. Next comes the main part of the filename. This can be from 1 to 8 characters long. If you wish you may add a period and 1 to 3 symbols after the main part of the filename.

APPEND
Format: APPEND filename

Description: The program on the disk is appended to the program in memory. The program on the disk must be in source code form (i.e., the SAVES command was used to place the program on the disk). Note that if two lines in the programs have the same linenumber, the line from the appended program will overwrite and eliminate the line in the current program.

Examples:

APPEND ALPHA appends the program named ALPHA on disk drive 1 to the current program.

APPEND D2:BETA appends the program named BETA on disk drive 2 to the current program.

COMPILE
Formats:

COMPILE	COMPILE filename/filename
COMPILE n	COMPILE filename n
COMPILE filename	COMPILE filename/filename n

(n must be 1, 2, or 3)

Description: Used to compile a program. If no filename is specified, the program is assumed to be in the computer. If a filename is specified, the system assumes that the program is too large to compile entirely in the computer. The system then brings parts of the program from the disk as needed and sends the compiled code back to the disk. This makes the compile significantly slower, but allows very large programs to be compiled. If filename/filename is used, the second filename determines the disk and name to which the output code is stored. If only one filename is

given, the system uses that filename with a .COD extension as the compiled code filename.

Examples:

COMPILE compiles the program in memory. The resulting code can be stored with a SAVEC command.

COMPILE 1 causes the system to delete each program line after it has been compiled. This almost doubles the size of a program which can be compiled entirely in memory. Be sure to save the program first.

COMPILE 2 When the program is executed, the BASIC is removed. This increases available memory by about 17K in a non XL and by 3K or 4K in an XL. When the program ends, the BASIC has to be reloaded.

COMPILE 3 The N=3 option is the same as N=1 combined with an N=2 compile.

COMPILE ALPHA.BAS compiles the program on disk 1 named ALPHA.BAS. The compiled code will be stored on disk 1 with the name ALPHA.COD.

COMPILE ALPHA.BAS/D2:ALPHA.COD 2 compiles the program on disk 1 named ALPHA.BAS. The compiled code is stored on disk 2 with the name ALPHA.COD. The BASIC is removed when the program is executed, allowing the maximum possible amount of memory.

DEL
Format: DEL linenumber,linenumber

Description: Deletes all linenumbers between and including the two listed. The first linenumber must be less than the second linenumber. To delete the special subroutines PUSING.APP and DLISTINT.APP use the command DEL 32768,65535.

DIR
Formats: DIR Dn: (n is the drive number and must be 1,2,3,or 4)
DIR
Description: Lists the directory for the specified disk. If no disk is specified, lists the directory for disk 1. Note that using DIR does not effect any program in memory. So if you are ready to save a program to a disk, you can use the DIR command to check the disk before you save the program.

Examples:

DIR lists directory of disk 1.

DIR D2: lists directory of disk 2.

EXEC
Formats: EXEC
EXEC 1
EXEC filename
Description: Executes a previously compiled program. If a filename is specified, the code is loaded from the file on the specified disk. Otherwise, the system assumes that a program in the computer has just been compiled. If a space and a one follow the EXEC command, any program in the computer will be deleted before execution begins. This increases the amount of memory available. Be sure to save the program before doing this. If the program is loaded from a file, any program in the computer will also be automatically deleted before the execution.

Examples:

EXEC
EXEC ALPHA.COD

KILL
Format: KILL filename

Description: Deletes a file from the specified disk.

Examples:

KILL ALPHA.DAT deletes the file ALPHA.DAT from disk 1.

KILL D2:BETA.D1 deletes the file BETA.D1 from disk 2.

LIST or L
Formats: LIST (or you can use just L but not L.)
LIST linenumber
LIST linenumber,linenumber
Description: LIST without linenumber lists the entire program. LIST followed by one linenumber lists only that line. LIST followed by two linenumbers lists all the lines between and including the two linenumbers; however, the first linenumber must be less than the second linenumber. You can use the abbreviation L for LIST, but not L followed by a period.

LLIST
Formats: LLIST
LLIST linenumber
LLIST linenumber,linenumber
Description: Works like LIST, except that the output is to the printer instead of the screen. See LIST and WIDTH.

LMARGIN
Format: LMARGIN number

Description: Sets left margin to value of number. Minimum is 0 and maximum is 39.

LOAD
Formats: LOAD filename
LOAD filename 1
Description: Loads a program which has been stored in token form (i.e., saved with SAVE command). The name and disk are specified in the filename. The second option (adding a space and 1) is used for programs too large to totally fit in the computer. Only a portion of the program is held in the computer; the remainder stays on the disk. This gives you a way to work with very large programs, because you can edit and add to the program as if it were entirely in the computer. See Chapter 11.

LOADS
Format: LOADS filename

Description: Loads a program which has been stored in non-token form.

Example: LOADS PHI.SR loads the program named PHI.SR stored on disk 1.

LOCK
Format: LOCK filename

Description: Sets the specified file to read only. Used to protect a file.

NEW
Format: NEW

Description: Deletes any previous program and prepares for the entry of a new program.

PEEK
Format: PEEK number

Description: Prints the value of the memory location specified by the number, which must be in decimal.

POKE
Format: POKE number,number

Description: Stores the value of the second number into the memory location specified by the first number. Be careful with this command. If you should change a key location, the BASIC will lock up and you will have to turn off the computer and reload.

RENAME
Format: RENAME filename/filename

Description: Changes the name of the file from the first (left) filename to the second (right) filename.

RUN
Formats: RUN RUN filename
RUN n RUN filename n (n=1, 2, or 3)
Description: Compiles and executes a program. If a filename is used, the program is loaded from the disk; otherwise, the program is assumed to be in the computer. The number serves as a command to the computer:

1--each program line is compiled and then deleted. This is used for programs which are too large to compile or execute in the available RAM. Be sure to save your program before using this option.

2--the BASIC is removed before program execution. This greatly increases the space available in a non-XL computer and gains several thousand bytes in an XL.

3--the BASIC is removed and the program lines deleted as they are compiled, providing the maximum amount of memory for program execution. Note that if the BASIC is removed, you will have to insert a disk with BASIC on it and reload the BASIC at the end of the program run. The system will issue a message telling you what to do.

Examples:

RUN compiles and executes the program in memory.

RUN 1 compiles (deleting program lines as they are compiled) and executes the program in memory.

RUN ALPHA loads, compiles, and executes the program named ALPHA on disk 1.

RUN D2:BETA 1 loads, compiles (deleting program lines as they are compiled), and executes the program named BETA on disk 2.

SAVE
Format: SAVE filename

Description: Saves the program in memory onto a disk using the name specified in filename. The program is stored in token form. If the disk already has a file with the same name, the first file will be destroyed.

Examples:

SAVE ALPHA saves a program onto disk 1 using the name ALPHA.

SAVE D2:ALPHA saves a program onto disk 2 using the name ALPHA.

Special note: The SAVE command should not be used with .COD or .WRK, because these names are used in some compile options and the program might be overwritten and lost.

SAVEC
Format: SAVEC filename

Description: Saves the compiled code to a disk using the name specified in filename. An EXEC command can then be used to execute this code without having to recompile it.

Example:

SAVEC CGCA saves the compiled code to disk 1 using the name CGCA.

SAVES
Format: SAVES filename

Description: Saves the program in memory onto a disk using the name specified in filename. The program is not stored in token form, and thus takes longer to load and uses more disk space than the SAVE command. The main reason to use SAVES is because you plan to append the program to another program.

Example:

SAVES GAMMA lists the program to the screen and simultaneously saves it in non-token form.

UNLOCK
Format: UNLOCK filename

Description: The LOCK command tells the DOS that a file is read only, thus protecting the file. The UNLOCK command removes this protection and allows the system to write to a file.

WIDTH
Format: WIDTH number

Description: Sets the printer width. The default value is 75. If you have a 40 column printer, you use the command WIDTH 40. Whatever width is set will remain until a new WIDTH command is given or the system is turned off.

Variables

Variables in Advan BASIC may be integer, real, or string. Integer variables must end with a % sign and string variables with a $ sign. All other characters must be capital letters, numbers, or periods. No spaces are permitted. The following are valid names:

Integers	Real	String
ALPHA%	TAX	NAME$
B%	AMOUNT	ADDRESS$
BETA.C2%	C	LAST.NAME$

All characters in a name are significant, including the % and $ symbols. Thus, NAME$ and NAME% are different variables. Appendix B contains the reserved words which may not be used as variables. Also note that because names starting with FN are reserved for functions, no variable names may begin with FN.

Integers have a maximum value of 32767 and a minimum value of -32768 (however, the smallest integer constant is -32767). Integer constants must end with a % sign. Each integer variable requires two bytes of memory space. The following are valid integer statements:

```
A%=B%+2%   C%=(C%+30000%)/(-3%)
```

Each real variable requires six bytes of memory and, depending on the number, nine or ten significant digits are held. The absolute value of a real number must be zero or greater than 10^-99 and less than 10^99. The following are valid real statements:

```
A=B+5     COUNT=(COUNT+3)/(-6)
```

Note that real and integer variables can be mixed in a statement. The program will have to make conversions, however, resulting in some loss of speed. These are legal expressions:

```
A=B%+2%     B%=(A+2%)/3
```

If integer and real variables and/or constants are mixed in an expression, the program will convert an integer to a real number when forced to do a numerical operation between them. If an integer variable is set equal to a real number, the real number is converted to an integer (it is rounded, not truncated) and the variable is set equal to the integer. For example:

```
10 A%=3.7
20 PRINT A%
RUN
4
```

String variables have a maximum length of 256 bytes. Unlike ATARI BASIC, strings in Advan BASIC do not have to be dimensioned.

Arrays

Arrays may be integer, real, or string. The maximum number of subscripts is 64. A DIM statement must appear in the program for a variable before it is used. The subscripts used in arrays may be integer or real numbers; however, the program will convert the real numbers to integers (rounding, not truncating), thus reducing execution speed.

Functions and Named Subroutines

User-defined one line functions are available. There may be 0 to 4 arguments and they may be integer, real, or string. All function names must start with FN. Functions may be integer, real, or string. They can be anywhere in the program and don't need to precede the use of the function. For example:

```
100 DEF FNA$(T$)=T$+".COD"
```

User-defined multi-line named subroutines are available. There may be 0 to 4 arguments and they may be integer, real, or string. All subroutine names must end with an @ symbol. See SUB command for more information.

Special note: In most cases the system will automatically convert between real and integer numbers. In user-defined functions and named subroutines, however, the argument must be the same in the definition and when the function or subroutine is used.

Operators

The following arithmetic operators are available for integers and real numbers: +, -, *, /, ^. For integers, the MOD operator causes a division and its result is the remainder. In the following example, 50 is divided by 8 and T% is set equal to the remainder.

```
100 T%=50% MOD 8%
110 PRINT T%
RUN
2
```

The plus sign may be used to concatenate strings. For example:

```
100 A$="ABC": B$="DEF"
110 C$=A$+B$: PRINT C$
RUN
ABCDEF
```

The following relational operators are available: >, >=, =, <=, <, <>. In the following line, if T% is greater than A%, the program will branch to line 100:

```
50 IF T%>A% THEN 100
```

Relation conditions can be combined using AND and OR operators. In the following line, if T% is greater than A%, or B1$ is less than NAME1$, the program will branch to line 100:

```
50 IF T%>A% OR B1$<NAME1$ THEN 100
```

If AND and OR operators are mixed in a condition, AND will be evaluated first.:

```
100 IF A%>5% OR B%>6% AND C5>7% THEN 200
```

The above line is equivalent to

```
100 IF A%>5% OR (B%>6% AND C%>7%) THEN 200
```

AND and OR can be used to perform binary operations on the bits of two integers;

```
10 T%=A% AND 1%
```

AND is performed between the binary bits of A% and those of 1%, and the resulting number placed in T%. For example, bit 0 of 1% is compared with bit 0 of A%; if both are one, a 1 is placed in bit 0 of T%. Otherwise 0 is placed in bit 0 of T%. The other bits of A% and 1% are compared in the same way. The above example has the interesting property that T% will equal 1 if A% is odd and zero if A% is even (a simple way to test whether a number is odd or even). Consider the next line:

```
10 T%=A% OR B%
```

This works like the previous example, except that here an OR is performed between the binary bits of A% and the binary bits of B% and the result placed in T%. For example, bit 5 of A% is compared with bit 5 of B%. If either bit is 1, then bit 5 of T% is set to 1; otherwise it is set to zero.

Special operators: +=, >>, and <<

There are many statements where something is added to a variable. For example:

```
T%=T%+3%        A%(5%)=A%(5%)+C%+2%
B$=B$+","       B(2%)=B(2%)+EXP(2)
C%=C%-1%
```

Using the += command, these statements can be rewritten in the following way, in many cases resulting in faster program execution:

```
T%+=3%          A%(5%)+=C%+2%
B$+=","         B(2%)+=EXP(2)
C%+=-1%
```

The >> and << commands are used only with integers. In the following line, the bits of A% are shifted three places to the right. For a positive number this is the same as dividing by 8 (that is, by 2^3).

```
10 T%=A%>>3%
```

In the next example, the bits of A% are shifted three places to the left. This is the same as multiplying by 8 (by 2^3).

```
10 T%=A%<<3%
```

Advan BASIC Commands

General Commands

ADR
[redacted]
[redacted]
DATA
[redacted]
DEG
DIM
END
[redacted]
FAST
FAST END
GOSUB
GOTO
[redacted]
LET
MACHINE
OFFDISPLAY
ONDISPLAY
PEEK
PEEKW
POKE
POKEW
[redacted]
RAD
READ
REM
RESTORE
RETURN
RTIME
SETARRAY
SUB
SUBEND
TIME
TRAP
WAIT

Decision

[redacted]
CASE END
CASE ELSE
&
IF THEN ELSE
IF DO ELSE ENDIF
ON GOSUB
ON GOTO

Loops

FOR NEXT STEP
REPEAT UNTIL
WHILE WEND

I/O

GETKEY
INPUT
INPUTLINE
LPRINT
LPRINT USING
PRINT
PRINT USING
STICK
STRIG
TAB

DISK

CLOSE
[redacted]
GET
NOTE
OPEN
POINT
PUT

Graphics/Sound

ASOUND
CINT@
COLOR
COLL
DFILL
DRAWTO
FILL
GRAPHICS
HPOS
LOCATE
ESCREEN
PCONTROL
PDISPLAY
PLOT
POS
PRATE
PSETCOLOR
PSIZE
SCONTROL
SETCOLOR
SETINT@
SOUND

Integer Functions

ABS%
ASC
ASCB
ASCW
RND%
VAL%

Real Functions

ABS
ATAN
COS
EXP
FINT
FIX
INT
LOG
RND
SIN
SQR
TAN
VAL

String Functions

CHR$
CHRW$
INSERTB
INSERTW
INSTR
INSTRI
LEFT
LEN
MID
NUM$
RIGHT
STRING
STR$

Note: In the following descriptions of commands, the terms integerexpression, realexpression, stringexpression, expression, and condition are used. These are valid integerexpressions:

```
A%    ALPHA2%    (T%+3%)/8%
```

These are valid realexpressions:

```
ALPHA    BETA+3    4    BETA^2+ALPHA+2%+C%
```

Note that integers can be used in real expressions; however, program execution will be slower because they have to be converted to real numbers.

These are valid stringexpressions:

```
ALPHA$    "ABC"    ALPHA$+BETA$+"A"
```

If the general term 'expression' is used, it includes any of the three types mentioned above.

Conditions are expressions with relational operators. These are valid conditions:

```
A%>B%    A$<="ABC"
```

Special Note: If a realexpression is used instead of an integerexpression or vice-versa, in almost all cases the system will accept the expression and convert it to the correct form.

ABS
Type: real function

Format: ABS(realexpression)

Description: Takes the absolute value of a real number.

Example:

```
10 Y=-5.2
20 PRINT ABS(Y),ABS(-3.12),ABS(2.1),ABS(3%)
RUN
5.2     3.12     2.1     3
```

ABS%
Type: integer function

Format: ABS%(integerexpression)

Description: Takes the absolute value of an integer.

Example:

```
10 T%=-3%
20 PRINT ABS%(T%),ABS%(-2%),ABS%(52%),ABS%(-2.7)
RUN
3     2       52      3
```

ADR
Type: integer function

Formats: ADR(variable)
ADR(linenumber)
Description: Returns an integer equal to the address of the specified variable or linenumber. Used mainly in machine code subroutines.

Example:

```
10 DIM A%(20%)
20 PRINT ADR(B%),ADR(C),ADR(D$)
30 PRINT ADR(A%(2%)),ADR(20)
```

ASC
Type: integer function

Format: ASC(stringexpression)

Description: Returns an integer equal to the ASCII code for the first character of the specified string.

Example:

```
10 A$="ABC"
20 PRINT ASC(A$),ASC("CD")
RUN
65      67
```

ASCB
Type: integer function

Format: ASCB(stringexpression,integerexpression)

Description: Returns the integer equal to the ASCII code of the Nth string character, where N equals the integerexpression.

Example:

```
10 A$="ABC":T%=2%
20 PRINT ASCB(A$,T%),ASCB("ABC",3%)
RUN
66      67
```

ASCW
Type: integer function

Format: ASCW(stringexpression,intexpresson)

Description: Returns an integer equal to the ASCII code of the Nth string character+256 times the ASCII code of the (N+1)th string character, where N equals the integerexpression.

Examples:

```
10 A$="AB"
20 PRINT ASCW(A$,1%)
RUN
16961
```

```
10 A$=CHRW$(62%)+CHRW$(312%)
20 PRINT ASCW(A$,1%),ASCW(A$,3%)
RUN
62      312
```

ASOUND
Type: BASIC command

Format: ASOUND integerexpression,ADR(linenumber)

Description: The integerexpression equals the voice number and must be 0, 1, 2, or 3. This command lets you set up a sequence of notes that the computer plays automatically; that is, the computer does not have to issue a new command for each note. ADR(linenumber) tells the compiler where the data is. The data determines the frequency, duration, amplitude, and distortion of each note. Note that the sound will not start until an SCONTROL command is given. This allows all four voices to be synchronized. See SOUND and Chapter 12 for a more complete discussion.

ATAN
Type: real function

Format: ATAN(realexpression)

Description: Calculates the arctan of the realexpression. The answer is in radians unless you have given the DEG command.

CASE, CASE ELSE, CASE END, &
Type: BASIC command

```
Format:  CASE condition
              (statements)
         & condition
              (statements)
         & condition
              (statements)
            .
            .
            .
         CASE ELSE
              (statements)
       CASE END
```

Description: CASE ELSE is optional. You can use as many & conditions as you like. If the condition following CASE is true, the statements between that condition and the & condition are executed and the program skips down to the statement following CASE END. If the condition following one of the &'s is true, the statements between that conditon and the next case-type (&, CASE ELSE, or CASE END) command are executed, and the program skips to the statement following CASE END. If none of the conditions are true, the program executes the statements following CASE ELSE if it is present; otherwise the program skips to the statement following CASE END. See Chapter 4.

CHR$
Type: string function

Format: CHR$(integerexpression)

Description: Generates a one character string. The integerexpression equals the ASCII code of the character generated.

Example:

```
10 B$=CHR$(65%)+CHR$(49%)
20 PRINT B$
RUN
A1
```

CHRW$
Type: string function

Format: CHRW$(integerexpression)

Description: Used mainly to save integers in strings. Generates a two character string. The first character equals the ASCII code for the remainder after the integerexpression is divided by 256. The second character equals the ASCII code for the integerexpression divided by 256.

Example:

```
10 A$=CHRW$(65%+256%*66%)
20 B$=CHRW$(515%)+CHRW$(5%)
30 PRINT A$,ASCW(B$,1%),ASCW(B$,3%)
RUN
AB    515     5
```

CINT@
Type: BASIC command

Format: CINT@ integerexpression,integerexpression

Description: Designed to work with ATARI's display list interrupt capability. SETINT@ specifies the screen line where an interrupt is to occur and also specifies a value and a location where the value is to be stored. CINT@ changes the value stored at the interrupt. The first integerexpression gives the identifying number of the interrupt (this is set by SETINT@) and the second integerexpression gives the new value. Because it changes the value stored without having to reset an interrupt, CINT@ is much faster than SETINT@. Note that you must append DLISTINT.APP before using this command. See SETINT@ and Chapter 15.

CLOSE
Type: BASIC command

Format: CLOSE integerexpression

Description: Ends disk operations for the channel number equal to the integerexpression. Note that when a program ends or the BREAK key is pressed, all open channels are automatically closed.

Example:

```
10 OPEN "I",1%,"T.DAT"
20 GET 1%,N%
30 CLOSE 1%
```

CODE
Type: BASIC command

Format: CODE"assembly language data"

Description: Allows assembly language code to be inserted into a program.

See Chapter 16 for specific data. Special assembly language mnemonics may be used. Note that all machine language code must be preceded by the command MACHINE followed by a linenumber. See MACHINE and CODEL.

Examples:

```
1000 MACHINE 1100
1010 CODE"LDA,F2,2,STA,DF,4,RTS"
1100 END
```

The number in memory location 2F2 is loaded into the accumulator and then stored in memory location 4DF. Note that the memory locations are in hex, and that all machine language code must end in RTS. Also the X register must not be altered by the machine language code. If you use the X register, you must first save it and then restore it before the RTS. If you want to use a decimal number in a CODE statement, it must not exceed 255 and it must be preceded by !.

```
100 MACHINE 200
110 CODE"LDAIM,!15,STA,DF,4,RTS"
```

Here, the decimal number 15 is loaded into the accumulator (LDAIM stands for LOAD immediate--see Appendix D for a list of mnemonics). The 15 is then stored in 4DF.

Linenumbers can be used in the CODE command. They must be preceded by @ if used with branch commands, or by # if used with JMP or JSR. In the following example, if the number in 2F2 is non-zero, the program goes to 120 and returns; otherwise 1 is loaded to the accumulator and stored in 2F2 before the program returns. The last example illustrates the use of the # symbol.

```
100 MACHINE 200
110 CODE"LDA,2F,2,BNE,@120,LDAIM,1,STA,2F,2"
120 CODE"RTS"
200 END
```

```
100 MACHINE 200
110 CODE"LDA,2F,2,BNE,@120,LDAIM,1,STA,DF,4,JMP,#130
120 CODE"LDAIM,0,STA,DF,4"
130 CODE"INC,2F,2,RTS"
200 END
```

CODEL

Type: BASIC command

Formats:

CODEL(variablename)	CODEL(variablename+"L")
CODEL(linenumber)	CODEL(variablename+"H")
CODEL(variablename+integer)	CODEL(linenumber+"L")
CODEL(linenumber+integer)	CODEL(linenumber+"H")

Description: Generates code for the address of a linenumber or a variable. If you use the +integer option (e.g., CODEL(T%+2)), the compiler adds the integer to the address of the linenumber or variable. If you use the +"L" or +"H" options, the compiler generates the low order or high order byte of

the address. For example, if the variable T% is at address 8E2, the CODEL(T%+"L") generates the hex byte E2. See MACHINE and CODE.

COLL
Type: integer function

Format: COLL(integerexpression,integerexpression)

Description: Returns an integer whose value depends upon whether or not collisions have occurred between players, missiles, or playfields. If the second integerexpression is zero, collisions with playfields are examined; if it is 16%, collisions with players are examined. The following chart shows how the value of the first integerexpression is used:

Value of first integerexpression	Collision examined with
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3

Adding 128 to the first integerexpression will clear the specified collision register.
Examples:

T%=COLL(0%,0%) returns an integer whose value depends upon whether or not player 0 has collided with the playfield.

T%=COLL(5%,0%) returns an integer whose value depends upon whether or not missile 1 has collided with the playfield.

T%=COLL(2%,16%) returns an integer whose value depends upon whether or not player 2 has collided with other players.

To determine which player or playfield the collision was with, you can AND the returned integer with 1%, 2%, 4%, or 8%. 1% corresponds to player or playfield 0, 2% corresponds to player or playfield 1, 4% to player or playfield 2, and 8% to player or playfield 3.

T%=COLL(2%,16%) AND 8% returns 1 if player 2 collided with player 3.

T%=COLL(5%,0%) AND 2% returns 1 if missile 1 collided with playfield 1.

T%=COLL(133%,0%) resets the collision register between missile 2 and the playfield (133=128+5).

Special note 1: The collision registers are not updated until the vertical blank occurs. Thus, there can be a delay between the resetting of a collision and when it is next set.

Special note 2: If the collision still exists, resetting the register will not clear it.

Special note 3: You should clear any collision registers that you plan to use at the start of a program.

COLOR
Type: BASIC command

Format: COLOR integerexpression

Description: The integerexpression determines the color which will be placed on the screen by any succeeding PLOT statements. The following example plots a point in the upper left portion of the screen; the color is determined by color register 1.

```
10 GRAPHICS 3%
20 COLOR 2%
30 PLOT 5%,3%
```

COS
Type: real function

Format: COS(realexpression)

Description: Returns the cosine of the value of the realexpression. Radians are assumed unless you have given the DEG command.

Example:

```
10 A=0.32
20 PRINT COS(A)
```

DATA
Type: BASIC command

Format: DATA data items

Description: Used with READ statements to enter data. DATA statements are non-executing and may be placed anywhere in the program, but must be the first statement on a line. See READ. If you want to put a comma in a string, use the inverse key to enter an inverted comma (a black comma in a white background). During the compile, Advan BASIC automatically switches inverted commas in data statements to normal commas.

Example:

```
10 READ A,A$,A%
20 PRINT A,A&,A%
30 DATA 5.2,ABC!,-5
RUN
5.2      ABC!      -5
```

DEF
Type: BASIC command

Formats: DEF FNvariablename
DEF FNvariablename(variablename,...,variablename)

See Chapter 8 for a detailed description. The variablename can be integer,string, or floating point, and there may be no more than four variablenames in parentheses.

DEG
Type: BASIC command

Format: DEG

Description: Causes the BASIC to assume degrees for all trig functions.

Example:

```
10 A=30
20 DEG
30 PRINT SIN(A)
```

DIM
Type: BASIC command

Format: DIM variablename(number),variablename,(number,...,number),...

Description: The variablename can be integer, real, or string. All arrays must be dimensioned, even those whose dimension is less than 10; all arrays are set to zero at the start of program execution. The DIM statement must precede the use of the array. If it doesn't, you will get two error messages: array redefined error at the DIM statement and argument error when you use the array. The number in the DIM statement gives the maximum value of the subscript. The minimum value is zero. The maximum number of subscripts is 64.

Special note: The DIM statement in Advan BASIC serves a completely different role for strings than in ATARI BASIC. Remember, in ATARI BASIC you do not have string arrays and DIM is used to indicate the length of a string. In Advan BASIC, however, you do not need to specify the length of a string, and dimensioning a string sets up a string array.

DFILL
Type: BASIC command

Format: DFILL integerexpression,integerexpression

Description: Used to fill the entire screen, player or missile with a particular value. The most common use is to clear the screen, player, or missile. The value of the first integerexpression determines what will be filled according to the following chart:

Value of 1st Integerexpression	Area filled
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3
16	entire screen

The value of the second integerexpression is stored in the filled area.

Examples:

DFILL 16%,0% clears the entire screen much faster than the GRAPHICS command does.

DFILL 0%,255% turns on all the data points of player 0; that is, player 0 will be a uniformly colored bar.

Special note: DFILL 16%,integerexpression can cause problems in mode 0 if you are using the editor to input data from the screen. This is because DFILL does not reset the cursor position nor some of the other variables used by the editor. You can rapidly clear the screen in mode 0 with the command PRINT CHR$(125%).

DRAWTO
Type: BASIC command

Format: DRAWTO integerexpression,integerexpression

Description: Draws a line from the last plotted point to the point whose column (horizontal position) equals the first integerexpression and whose vertical position equals the second integerexpression. The color of the line is determined by the last color command. Note that this command does not work in mode 0. The following program draws a line on the screen and then waits about 4 seconds before it clears the screen and returns to the text mode.

```
100 GRAPHICS 3%
110 COLOR 2%
120 PLOT 2%,3%
130 DRAWTO 8%,3%
140 WAIT 240%
150 END
```

END
Type: BASIC command

Format: END

Description: Stops program execution and returns control to BASIC. Note

that END can appear anywhere in the program. In fact, several ENDs may be in a program. The compiler always inserts an END after the last program statement. The following program inputs numbers and prints the numbers until -1 is entered, which returns control to BASIC.

Example:

```
100 INPUT A%
110 IF A%=-1% THEN END
120 PRINT A%: GOTO 100
```

EOF
Type: integer function

Format: EOF(integerexpression)

Description: Tests whether you are at the end of a disk file. The integerexpression equals the file number and must be 0, 1, 2, or 3. The function returns 1 if you are at the end of the file and 0 if not. The following program opens the disk file named DATA, which is located on disk 1. It reads and prints all the strings from the file.

Example:

```
100 OPEN "I",1%,"DATA"
110 IF EOF(1%)=0% THEN GET 1%,A$: PRINT A$: GOTO 110
120 CLOSE 1%: END
```

EXG
Type: BASIC command

Format: EXG(stringvariable,stringvariable)

Description: Exchanges the two strings.

Example:

```
100 A$="ABC": B$="ZYXW"
110 EXG(A$,B$)
120 PRINT A$,B$
130 END
RUN
ZYXW    ABC
```

EXP
Type: real function

Format: EXP(realexpression)

Description: Calculates the value of e^x where x equals the value of the realexpression.

Example:

```
100 X=3
110 PRINT EXP(X+1)
120 END
```

FAST and FAST END
Type: BASIC command

Format:
```
FAST
 .
 .
 .
FAST END
```

Description: For the parts of the program between FAST and FAST END, Advan's optional optimizing compiler produces machine language code, which is faster but takes up more room than pseudo code. The optimizing compiler also reduces the length of the pseudo code regions (those not bracketed by FAST-FAST END) by about 20 to 25%. Because only about 10% of many programs is speed critical, the optimizing compiler can often significantly improve speed without changing program length. These commands are ignored by the standard Advan BASIC.

FILL
Type: BASIC command

Format: FILL integerexpression,integerexpression

Description: Draws a line from the last plotted point to the point whose column (horizontal position) equals the first integerexpression and whose vertical position equals the second integerexpression (where 0 equals the top line). As each point in the line is drawn, the system fills in empty points to the right of the point until the screen edge or a plotted point is encountered. Note that FILL does not work in mode 0.

Example:

```
10 GRAPHICS 3%
20 COLOR 2%
30 PLOT 5%,10%
40 DRAWTO 10%,15%
50 DRAWTO 0%,15%
60 FILL 4%,11%
```

The point at 5,10 is plotted. Line 40 draws a line from 5,10 to 10,15. Line 50 draws a line from 10,15 to 0,15. The FILL command draws a line from 0,15 to 4,11 and fills in the triangle as it draws the line.

FINT
Type: Real function

Format: FINT(integerexpression)

Description: Integers can take on values only from -32768 to 32767. If you consider the integers as unsigned numbers, the range is 0 to 65535. In some cases, such as memory locations, unsigned integers are more helpful, FINT assumes the integer to be an unsigned number and converts it to a floating point number for printing or testing.

Example:

```
100 FOR T%=50000% TO 50004%
110   PRINT T%,FINT(T%)
120 NEXT T%
RUN
-15536  50000
-15535  50001
-15534  50002
-15533  50003
-15532  50004
```

FIX
Type: Real function

Format: FIX(realexpression,integerexpression)

Description: Returns the real number specified by realexpression rounded to the number of decimal points specified by integerexpression.

Example:

```
10 T=FIX(4.372,2%):PRINT T
RUN
4.37
```

FOR NEXT STEP
Type: BASIC command

Format: FOR variablename=expression TO expression STEP expression

```
          .
          .
          .
        NEXT variablename
```

Description: Used to loop through a sequence of statements a fixed number of times. Variablename must be an integer or a real number; it must not be an array element. The variablename with FOR must match the variablename with NEXT. STEP is optional and if not present, the third expression is assumed to be 1. All three expressions may be either real or integer; however, the program runs faster if they are the same type as the variablename.

GET
Type: BASIC command

Format: GET integerexpression,variablename

Description: Reads data from a file. The file number equals the integerexpression. The variablename can be integer, real, or string. See PUT. The following program reads and prints a string, an integer, and then two real numbers from a file called "DATA.FIL". See Ch. 6 for alternate form of GET.

Example:

```
100 OPEN "I",1%,"DATA.FIL"
110 GET 1%,A$: GET 1%,N%: GET 1%,B: GET 1%,C
120 PRINT A$,N%,B,C
130 CLOSE 1%
140 END
```

GETKEY
Type: integer function

Format: GETKEY

Description: Returns ASCII code (See Appendix A) for character entered at keyboard. Returns zero if no character, or if you already used GETKEY to get that character.

GOSUB
Type: BASIC command

Format: GOSUB linenumber

Description: Transfers control to a subroutine located at the specified linenumber. When the program reaches a RETURN, control returns to the statement immediately after the GOSUB.

GOTO
Type: BASIC command

Format: GOTO linenumber

Description: Transfers control to the specified linenumber.

GRAPHICS
Type: BASIC command

Format: GRAPHICS integerexpression

Description: Sets the graphics mode. See Chapter 13 for a complete discussion. There are 16 graphics modes (0 through 15). Many of the modes can be opened with or without a text window at the bottom of the screen. Also you can have graphics with or without player-missiles and with or

without alternate character sets. First you must pick your graphics mode. (See Table 13-1, Ch. 13) If you don't want a text window, add 16. If you don't want the screen cleared when you open the display, add 32. If you want player-missiles, add 64. If you want an alternate character set, add 128.

Example:

```
100 GRAPHICS 84
```

84 equals 4+16+64, so this command produces graphics mode 4 with no text window and with player-missiles.

HPOS
Type: BASIC command

Format: HPOS integerexpression,integerexpression

Description: Sets the horizontal position of a player or missile. The first integerexpression determines which player or missile is set according to the following chart.

Value of first Integerexpression	Player-missile
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3

The value of the second integerexpression determines the horizontal position. 40 places the player or missile near the left screen edge and 216 near the right screen edge.

IF DO ELSE ENDIF
Type: BASIC command

Format:
```
IF condition DO
   .
   .
   .
ELSE
   .
   .
   .
ENDIF
```

(Note that ELSE is optional)

Description: This is a multi-line version of IF THEN ELSE. If ELSE is used and the condition is true, the statements between DO and ELSE are executed and those between ELSE and ENDIF are skipped. If the condition is false, the statements between ELSE and ENDIF are executed and those between DO and ELSE are skipped. If ELSE is not used, the statements between DO and ENDIF are executed if the condition is true, and skipped if the condition is false. You may use as many lines as necessary between DO and ELSE and between ELSE and ENDIF. Also you may nest IF DO ELSE ENDIF.

IF THEN ELSE
Type: BASIC command

Formats: IF condition THEN statement(s) ELSE statement(s)

IF condition THEN linenumber ELSE statement(s)

IF condition THEN statement(s) ELSE linenumber

IF condition THEN linenumber ELSE linenumber

(Note that ELSE is optional in all cases.)

Description: If the condition is true, the statements after ELSE are skipped and those after THEN executed. If the condition is false, the statements after THEN are skipped and those after ELSE are executed.

INPUT
Type: BASIC command

Formats: INPUT variablename
INPUT "message"variablename,...,variablename

Description: Allows data to be entered from the keyboard and assigned to a variable. For the first option a question mark is printed. If the second option is used, the message is printed with no question mark unless there is one in the message. (See Ch. 3)

INPUTLINE
Type: BASIC command

Formats: INPUTLINE stringvariable
INPUTLINE "message"stringvariable

Description: Inputs a line of string data from the keyboard. The final carriage return will not be a part of the string. The advantage of this command over INPUT is that strings with commas can be input. In the INPUT command, a comma means the end of the string. In the first option, a question mark is printed. In the second option, the message is printed without a question mark, unless one is included in the message. See Chapter 3.

INSERTB
Type: BASIC command

Format: INSERTB(stringvariablename,integerexpress,integerexpress)

Description: Inserts a byte into the string specified by the stringvariablename. The byte inserted equals the value of the second integerexpression, and its position in the string is determined by the value of the first integerexpression. In the following example, the first integerexpression equals 3 and the second integerexpression equals 65 (ASCII code for A), and thus, the third character of the string is set to 65.

Example:

```
100 A$="ABCD"
110 INSERTB(A$,3%,65%)
120 PRINT A$
RUN
ABAD
```

INSERTW
Type: BASIC command

Format: INSERTW(stringvariablename,interexpress,integerexpress)

Description: Inserts a word (two bytes) into a string. The word inserted equals the value of the second integerexpression and its location is determined by the value of the first integerexpression. In the following example, the second integerexpression equals 65+66*256 and the first integerexpression equals 2. Thus, the second character of the string is set to 65 (the low order part of the word) and the third character of the string is set to 66 (the high order part of the word). Note that 65 is the ASCII code for A and 66 for B.

Example:

```
100 A$="ZZZZ"
110 INSERTW(A$,2%,65%+66%*256%)
120 PRINT A$
RUN
ZABZ
```

INSTR
Type: integer function

Format: INSTR(integerexpression,stringexpression,stringexpression)

Description: Searches the first stringexpression for a match with the second stringexpression. The value of integerexpression determines the position in the string where the search begins. Because INSTR1 is much faster than INSTR, it should be used wherever possible (see INSTR1). In the first example, INSTR starts at the second character searching for the string "CD". At the third character, it finds a match, and so the function returns a 3. If INSTR does not find a match, it returns zero.

Examples:

```
100 A$="ABCDE"
110 PRINT INSTR(2%,A$,"CD"),INSTR(1%,A$,"D")
120 END
RUN
3   4


100 A$="AB,DE,FG": T$=",F"
110 PRINT INSTR(1%,A$,T$),INSTR(3%,A$,"B")
120 END
RUN
6     0
```

INSTR1
Type: integer function

Format: INSTR1(stringexpression,integerexpression,integerexpression)

Description: Searches the stringexpression for the byte whose value equals the second integerexpression. It starts at the string location equal to the first integerexpression. At line 110 in the following example, the function starts at the second location in the string and searches for 69 (the ASCII code for E). It finds 69 in the fifth string location and returns 5. If the byte is not found, it returns zero.

Example:

```
100 A$="ABCDE"
110 PRINT INSTR1(A$,2%,69%)
120 PRINT INSTR1(A$,4%,ASC("C"))
130 END
RUN
5
0
```

INT
Type: real function

Format: INT(realexpression)

Description: Returns the integer part of a real number. Thus, INT(4.7) equals 4 and INT(4) equals 4. A negative number is treated as a negative integer plus a positive fraction, and the negative integer is returned. For example, -16.2 equals -17+.8 and so the function returns -17.

Example:

```
100 PRINT INT(2),INT(6.8),INT(-5),INT(-6.4)
RUN
2     6      -5     -7
```

LEFT
Type: string function

Format: LEFT(stringexpression,integerexpression)

Description: Returns a string equal to the left part of the string, starting with the first character and including the character at the string location equal to the integerexpression. If the integerexpression equals zero, a zero length (null string) is returned, as in the example below.

Example:

```
100 A$="ABCDEF"
110 PRINT LEFT(A$,3%)
120 PRINT LEFT(A$,0%)
130 END
RUN
ABC
```

LEN
Type: integer function

Format: LEN(stringexpression)

Description: Returns an integer equal to the length of the stringexpression.

Example:

```
100 A$="ABC": B$="ZZ"
110 PRINT LEN(A$),LEN"CD",LEN(A$+B$)
120 END
RUN
3       2       5
```

LET
Type: BASIC command

Format: LET variable=expression

Description: Assigns the value of the expression to the variable. Note that LET is optional.

Example:

```
100 LET A$="ABC"
110 B$="ABC"
```

LOADST
Type: BASIC command

Format: LOADST(expression)

Description: Loads the value of the expression to the stack. (The BASIC stack, not the hardware stack). The expression can be integer, real, or string. LOADST and POPST are normally used to pass a value to a subroutine or to return a value from a subroutine. They can also be used to make subroutines recoursive. See POPST.

Examples:

```
100 T%=5%: LOADST(T%+2%)
110 GOSUB 200: POPST(T$)
120 PRINT T$
130 END
200 POPST(Y%)
210 IF Y%=12% THEN LOADST("DEC"): ELSE LOADST("NOT DEC")
220 RETURN
```

```
100 HEX@(T%): POPST(T%)
110 PRINT CHR$(T%)
120 END
130 SUB HEX@(X%)
140 IF X%<10% THEN LOADST(48%+X%) ELSE LOADST(55%+X%)
150 SUBEND
```

LOCATE
Type: integer function

Format: LOCATE(integerexpression,integerexpression)

Description: Returns the value of the display point at the screen location specified by the integerexpressions. The first is the column number of the point (far left column is 0), and the second is the vertical line number (top line is 0).

LOG
Type: real function

Format: LOG(realexpression)

Description. Returns the natural log of the realexpression.

LPRINT
Type: BASIC command

Format: LPRINT expression,...,expression

Description: The same as PRINT, except the output is to the printer instead of the screen (see PRINT).

LPRINT USING
Type: BASIC command

Format: LPRINT USING stringexpression,expression,...,expression

Description: The same as PRINT USING, except the output is to the printer instead of the screen (see PRINT USING).

LSCREEN
Type: BASIC command

Format: LSCREEN integerexpression,stringexpression

Description: Loads the display screen with data from the disk file whose name is given by the stringexpression. The filenumber is set equal to the integerexpression and must be 0, 1, 2, or 3. At the end of the load, the file will be closed automatically. This command is designed to work with the optional screen design package.

MACHINE
Type: BASIC command

Format: MACHINE linenumber

Description: Normally the BASIC compiler generates pseudo code, which is executed by the execution module. There are situations demanding absolute maximum speed, which is only possible with assembly language code. Although Advan's optimizing compiler can compile to machine code, carefully constructed assembly language routines will usually do better. The MACHINE command lets you insert assembly language code into a program.

The code following the MACHINE command is assumed to be an assembly language program. It must end with RTS to return control to the BASIC. The CODE and CODEL commands are used to enter the assembly language program. Note that the X register must not be changed by the assembly language code. If you plan to use the X register, first save it (TXA,PHA is a good technique) and then reload it (PLA,TAX works with the above). When the program returns from the assembly language code, execution continues at the line given by the linenumber in the MACHINE command. See CODE, CODEL, and Chapter 16 for more information.

MID
Type: string function

Format: MID(stringexpression,integerexpression,integerexpression)

Description: Returns a string equal to a substring of the stringexpression. The value of the second integerexpression determines the length of the substring. The value of the first integerexpression determines the character location of the start of the substring.

Example:

```
100 A$="ABCDEF"
110 PRINT MID(A$,4%,2%)
120 END
RUN
DE
```

NOTE
Type: BASIC command

Format: NOTE integerexpression,integer variable,integer variable

Description: Used with POINT to set the the location in the file. The integerexpression determines the file number and must be 0, 1, 2, or 3. The system stores the sector number in the first integer variable and the byte position in the sector in the second integer variable. See POINT. The following program lets you get and print any one of a thousand strings quickly. Without POINT and NOTE, you would probably have to start at the beginning of the file each time and get each string until you came to the one you wanted. This would be much slower and cause more disk wear and tear.

Example:

```
100 OPEN"I",1%,"DATA.FIL"
110 DIM STRL%(1000,1)
120 FOR T%=1% TO 1000%
130  NOTE 1%,STRL%(T%,0%),STRL%(T%,1%)
140  GET 1%,C$
150 NEXT T%
180 INPUT"Enter string #" T%
190 IF T%=0% THEN END
200 POINT 1%,STRL%(T%,0%),STRL%(T%,1%)
210 GET 1%,C$
220 PRINT C$
230 GOTO 180
```

NUM$
Type: string function

Format: NUM$(integerexpression)

Description: Converts the integer given by the integerexpression into a string. For example, if the integerexpression equals 100, then NUM$ returns a string whose first byte is 49 (ASCII code for 1), and whose second and third bytes equal 48 (ASCI code for 0). See STR$.

OFFDISPLAY
Type: BASIC command

Format: OFFDISPLAY

Description: Turns display off, increasing program speed by up to 30%. ONDISPLAY turns display back on.

ONDISPLAY
Type: BASIC command

Format: ONDISPLAY

Description: Turns the display on, if the display has been blanked with

OFFDISPLAY.

ON GOSUB
Type: BASIC command

Format: ON integerexpression GOSUB linenumber,...,linenumber

Description: Acts like a GOSUB to one of the linenumbers. The value of the integerexpression determines which linenumber is used. For example, if integerexpression equals 3, the program executes a GOSUB to the third linenumber in the list. You will get an error message if the integerexpression is less than one or greater than the number of linenumbers in the list. When a RETURN is reached in the subroutine, the program returns to the statement following the ON GOSUB. See Chapter 4.

ON GOTO
Type: BASIC command

Format: ON integerexpression GOTO linenumber,...,linenumber

Description: Acts like a GOTO to the linenumber specified by the integerexpression. For example, if the integerexpression equals 2, the program executes a GOTO to the second linenumber in the list. You will get an error message if the integerexpression is less than one or greater than the number of linenumbers in the list. See Chapter 4.

OPEN
Type: BASIC command

Format: OPEN stringexpression,integerexpression,stringexpression

Description: Used to access a disk file. The integerexpression sets the filenumber amd must be 0, 1, 2, or 3. All subsequent commands which access this file must use this filenumber. The second stringexpression gives the disk filename (1 to 8 characters) and an optional extension of a period and 1 to 3 characters. If the disk is not disk 1, the name is preceded by D2:, D3:, or D4:. The following are legal names: DATA.FIL DATA D2:DATA.F2 D3:SAVEDATA.FLE

The first stringexpression determines the mode in which the file is to be opened (O, I, A, or R). Mode O opens the file for output only. If a file of the same name already exists on the disk, the file will be destroyed. Mode A opens the file in the append mode, and lets you append data to the end of the file. Mode I opens the file in input only mode. Mode R is the random access mode. Note that you cannot add to a file in R mode.

Example: OPEN "I",2%,"D2:ALPHA.C"

PCONTROL
Type: BASIC command

Format: PCONTROL integerexpr,integereexpr,integerexpr,integerexpr

Description: Used with PDISPLAY and PRATE, which define the figure and its rate of motion. PCONTROL starts and stops the motion, allowing synchronization of several players and missiles. The first integerexpression controls player and missile 0, the second integerexpression controls player amd missile 1, the third controls 2, and the fourth controls 3. The following chart shows how the value of the integerexpression controls player-missile action:

Value of Integerexpression	Effect
0	player and missile stop
1	player starts and missile stops
2	missile starts and player stops
3	both player and missile start

Example: 100 PCONTROL 3%,0%,1%,2% activates player 0, player 2, missile 0, and missile 3.

PDISPLAY
Type: BASIC command

Format: PDISPLAY integerexpress,ADR(linenumber),integerexpress

Description: Draws a figure into a player or missile. The first integerexpression determines which player or missile is being set according to the following chart:

Value of first Integerexpression	Player-missile
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3

The second integerexpression sets the vertical position of the player-missile. 128 is about the center of the screen. The linenumber specifies the location of the data to be stored in the player or missile (See Ch. 14).

PEEK
Type: integer function

Format: PEEK(integerexpression)

Description: Returns the value of the byte at the memory location specified by the integerexpression. Note that the integerexpression is assumed to be a positive number from 0 to 65535. See POKE and PEEKW.

Example:

100 PRINT PEEK(40960%) prints the number stored in memory location 40960.

PEEKW
Type: BASIC function

Format: PEEKW(integerexpression)

Description: Returns the value of the integer word (2 bytes long) at the memory location specified by the integerexpression. Note that the integerexpression is assumed to be a positive number from 0 to 65535. The low order part of a word is at the specified memory address, and the high order part is at the memory address plus 1.

Example:

```
100 PRINT PEEKW(40960%)
110 PRINT FINT(PEEKW(40960%))
```

Line 100 prints a number equal to the value in memory location 40969 plus 256 times the value of the number in location 40961. If this number is greater than 32767, it will be printed as a negative number. The FINT function in line 110 converts the integer returned by PEEKW to a positive floating point number. For example, if the number stored at 40960 is 38000, line 110 prints 38000 instead of treating the number like a signed integer (i.e., printing a negative number).

PLOT
Type: BASIC command

Format: PLOT integerexpression,integerexpression

Description: Plots a point or a character to the display screen. The horizontal position is determined by the first integerexpression, and the vertical position by the second integerexpression. The character or color plotted is determined by the COLOR command.

POINT
Type: BASIC command

format: POINT integerexpression,integerexpression,integerexpression

Description: Used with NOTE to set the location in the file. The first integerexpression determines the file number and must be 0, 1, 2, or 3. The second integerexpression determines the sector number, and the third integerexpression the byte position in the sector. See Chapter 6.

POKE
Type: BASIC command

Format: POKE integerexpression,integerexpression

Description: Stores the number specified by the second integerexpression into the memory location specified by the first integerexpression. Note that since a memory byte has a maximum value of 255 and a minimum value of 0, the second integerexpression should remain within these limits. You should realize that the POKE command can cause the system to crash. If you POKE a location used by BASIC, weird and bad things will probably happen.

Example:

100 POKE 40960%,3% causes 3 to be stored in memory location 40960.

POKEW
Type: BASIC command

Format: POKEW integerexpression,integerexpression

Description: Stores the two byte word specified by the second integerexpression into the memory locations specified by the first integerexpression. See PEEKW.

Example:

100 POKEW 40960%,515% stores 515 in two memory locations. 515 equals 2 times 256 plus 3. 3 is stored in 40960 and 2 in 40961.

POPST
Type: BASIC command

Format: POPST(variablename)

Description: Removes an integer, real number, or string from the stack and stores it in the variable. See LOADST.

POS
Type: BASIC command

Format: POS integerexpression,integerexpression

Description: Most useful for text modes. Positions cursor to column equal to the first integerexpression and the line equal to the second integerexpression. A subsequent PRINT will start printing at the new cursor position.

PRATE
Type: BASIC command

Format: PRATE integerexpr,integerexpr,integerexpr,integerexpr

Description: Used with PDISPLAY and PCONTROL to automatically move a player-missile. The first integerexpression determines the player-missile according to the following chart.

Value of first Integerexpression	Player-missile
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3

The second integerexpression determines the horizontal speed of the player-missile. A negative number moves the object to the left and a positive number to the right. The third integerexpression determines the vertical speed. A negative number moves the object up and a positive number moves it down. The maximum speed is 32767; however, speeds of around 256 are more reasonable. Due to wrap around, very high speeds cause weird effects. The fourth integerexpression determines the length of time a given figure is displayed in sixtieths of a second.

Example:

```
100 PRATE 2%,256%,512%,4%
```

sets player 2 motion to the right and down. Each figure will remain on the screen for 4/60 second. You can start or stop player or missile motion with PRATE or with PCONTROL. See Chapter 14 for more information.

PRINT
type: BASIC command

Format: PRINT expression,...,expression

Note that the commas can be replaced by semicolons

Description: Each of the expressions is evaluated and the result is output to the display screen. If a comma separates two expressions, the second expression will be shown in the next print zone. The first print zone starts at the left margin (usually print position 0, 1, or 2). The second print zone starts at position 8, the third at 16, etc. If a semicolon separates two expressions, the second will be output immediately following the first. If the last expression is not followed by a comma or semicolon, a carriage return will be output and the next PRINT will start on a new line. If the last expression is followed by a comma or semicolon, the next PRINT will be on the same line.

Examples:

```
100 PRINT "ABC"
110 PRINT "ZZ"
120 END
RUN
ABC
ZZ
```

```
100 PRINT "ABC",
110 PRINT "ZZ"
120 END
RUN
ABC     ZZ

100 PRINT "ABC";
110 PRINT "ZZ"
120 END
RUN
ABCZZ

100 FOR T%=1% TO 5%
110    PRINT "A";T%;
120 NEXT T%: PRINT: PRINT "DONE"
RUN
A1A2A3A4A5
DONE
```

PRINT USING
Type: BASIC command

Format: PRINT USING stringexpression,expression,...,expression

Description: The following examples illustrate the use of PRINT USING.

Examples: Note that the decimal points are aligned and that only two numbers are printed after the decimal point. Note also that the numbers are rounded. Each of the # signs in the stringexpression reserves a space for a digit of the number:

```
100 FOR T%=1% TO 3%
110    READ A
120    PRINT USING "###.##",A
130 NEXT T%
140 DATA 5.376,15.1,-17.312
RUN
   5.38
  15.10
-17.31
```

If the # signs in the stringexpression are preceded by $$, a single $ is placed immediately before the number:

```
100 FOR T%=1% TO 4
110    READ A
120    PRINT USING "$$###.##",A
130 NEXT T%
140 DATA 1.777,21.772,0.76,310.1
RUN
   $1.78
  $21.77
    $.76
$310.10
```

If the # signs in the stringexpression are preceded by **, any leading spaces in the number are filled with *'s:

```
100 FOR T%=1% TO 4%
110    READ A
120    PRINT USING "**##.##",A
130 NEXT T%
140 DATA 1.777,21.772,310.1,1107.666
RUN
***1.78
**21.77
*310.10
1107.67
```

If the # signs in the stringexpression are preceded by **$, a $ is placed immediately before the number and any leading spaces are filled with *'s:

```
100 FOR T%= 1% TO 4%
110    READ A
120    PRINT USING "**$#.##",A
130 NEXT T%
140 DATA 1.777,21.772,.076,310.1
RUN
**$1.78
*$21.77
***$.08
$310.10
```

If a minus sign follows the last # sign in the stringexpression, a trailing minus sign is printed for negative numbers. If a plus sign follows the last # sign in the stringexpression, a trailing plus or minus sign is printed, depending on the sign of the number. If a plus sign precedes the first # sign in the stringexpression, a plus or minus sign precedes the number when it is printed. Note that you cannot use the last option if you are using *'s or $'s:

```
100 FOR T%=1% TO 3%
110    READ A
120     PRINT USING "**#.##-  $$#.##+  +###.##",A,A,A
130 NEXT T%
140 DATA -1.777,21.772,-.076
RUN
**1.78-     $1.78-      -1.78
*21.77     $21.77+     +21.77
***.08-      $.08-       -.08
```

PRINT USING can also work with strings. If the stringexpression equals !, the first character of the string is printed:

```
100 PRINT USING "!","ABC"
RUN
A
```

If the stringexpression is a backslash followed by a backslash, two or more characters from the string are printed. The number printed equals two plus the number of spaces between the backslashes. If the string is shorter than the number of characters to be printed, spaces will be added

to the end of the string until it equals the number of characters to be printed. If the string is longer than the number of characters to be printed, the left most characters of the string will be printed:

```
100 FOR T%=1% TO 4%
110    READ A$
120    PRINT USING "\  \",A$
130 NEXT T%
140 DATA A,ABC,ABCDE,ABCDEF
RUN
A
ABC
ABCD
ABCD
```

The stringexpression can have several different print formats. Note that spaces in the stringexpression correspond to spaces in the output:

```
100 FOR T%=1% TO 2%
110    READ A,A$,B
120    PRINT USING "#.# ! ##",A,A$,B
130 NEXT T%
140 DATA 5,ABC,8.6,2.12,ZZ,-2
RUN
5.0 A  9
2.1 Z -2
```

If you have more expressions than formats in the stringexpression, the program will return to the start of the stringexpression:

```
100 A=2:A$="ABC": B=4: B$="Z"
110 PRINT USING "# \ \",A,A$,B,B$,A,A$
RUN
2 ABC4 Z  2 ABC
```

Special note: You must append PUSING.APP in order to compile a program which uses PRINT USING. Otherwise you will get a missing line error.

PSETCOLOR
Type: BASIC command

Format: PSETCOLOR integerexpress,integerexpress,integerexpress

Description: Sets the color and brightness for a player and its associated missile. The first integerexpression equals the player number and must be 0, 1, 2, or 3. The second integerexpression sets the color, and the third integerexpression sets the brightness. Chapter 13 describes the relationship between these numbers and the colors produced. Note that during the load of the BASIC, all players have their colors set to black, so you must always issue a PSETCOLOR command before using players or missiles. See SETCOLOR.

PSIZE
Type: BASIC command

Format: PSIZE integerexpression,integerexpression

Description: The second integerexpression sets the player-missile size. 0 is normal size, 1 is two times normal, and 3 is four times normal. The following chart shows how the first integerexpression sets the player-missile.

Value of first Integerexpression	Player-missile
0	player 0
1	" 1
2	" 2
3	" 3
4	missile 0
5	" 1
6	" 2
7	" 3

Example: PSIZE 2%,3% sets player 2 to four times normal size.

PUT
Type: BASIC command

Format: PUT integerexpression,variablename

Description: Places data into a file. The integerexpression equals the filenumber and must be 0, 1, 2, or 3. The variable can be integer, real or string. The maximum length of a string is 255 bytes. See GET. The following program puts a string, an integer, and then two real numbers into a file called "DATA.FIL". See Chapter 6 for an alternate form of PUT.

Example:

```
100 OPEN "O",0%,"DATA.FIL"
110 PUT 0%,A$: PUT 0%,N%: PUT 0%,B: PUT 0%,C
120 CLOSE 0%
130 END
```

RAD
Type: BASIC command

Format: RAD

Description: Causes BASIC to assume radians for all trig functions.

READ
Type: BASIC command

Format: READ variable,...,variable

Description: Gets data from DATA statements and assigns these values to the variables. See DATA and RESTORE.

Example:

```
100 READ A,A$
110 READ B$,A%,C$
120 PRINT A,A$
130 PRINT B$,A%,C$
140 DATA 5,ABC,&2
150 DATA 4,ZZ
RUN
5     ABC
&2    4      ZZ
```

REM
Type: BASIC command

Format: REM comments

Description: Provides a way to put comments and information into a program. When Advan BASIC encounters a REM, it goes on to the next statement. Unlike ATARI BASIC, Advan REM statements have no effect on program execution speed.

REPEAT UNTIL
Type: BASIC command

Format:
```
REPEAT
  .
  .
  .
UNTIL condition
```

Description: Provides a way of looping through a sequence of statements. The following program executes the statements between REPEAT and UNTIL and then tests the condition. If true, the program continues with the statement immediately following UNTIL. If false, it goes back to the REPEAT and again executes the statements between REPEAT and UNTIL. FOR/NEXT is a more useful looping tool if you know how many times a loop must be executed; otherwise REPEAT UNTIL and WHILE WEND are probably better.

Example:

```
100 REPEAT
110    READ A
120    S=S+A
130 UNTIL A=0
140 PRINT S
150 DATA 1,2,3,0
160 END
RUN
6
```

RESTORE
Type: BASIC command

Format: RESTORE

Description: As the program executes READ commands, it moves sequentially through the data in the DATA statements. RESTORE causes the program to return to the start of the data. The next READ statement reads the data from the start of the first DATA statement. See READ and DATA.

Example:

```
100 READ A%,B%: PRINT A%,B%
110 READ C%,D%: PRINT C%,D%
120 RESTORE
130 READ E%,F%: PRINT E%,F%
140 DATA 1,2,3,4,5,6
RUN
1      2
3      4
1      2
```

RETURN
Type: BASIC command

Format: RETURN

Description: Used with GOSUB. When the program executes a RETURN statement, control transfers to the statement right after the last executed GOSUB. See GOSUB.

RIGHT
Type: string function

Format: RIGHT(stringexpression,integerexpression)

Description: Returns a string equal to the right part of the stringexpression starting at the location equal to the value of the integerexpression.

Example:

```
100 A$="ABCDEF"
110 PRINT RIGHT(A$,4%)
RUN
DEF
```

RND
Type: real function

Format: RND(realexpression)

Description: Returns a random number less than the value of realexpression and greater than zero. Note that RND% is much faster than RND, although not as random. See RND%.

Example: 100 PRINT RND(8)

RND%
Type: integer function

Format: RND%(integerexpression)

Description: Returns a random integer less than or equal to the integerexpression and greater than or equal to one. The integerexpression must be less than or equal to 255. See RND.

Example:

```
100   PRINT RND%(6%)
```

RTIME
Type: BASIC command

Format: RTIME

Description: Sets to zero the clock used by the TIME and WAIT commands. See TIME and WAIT.

SCONTROL
Type: BASIC command

Format: SCONTROL integerexpr,integerexpr,integerexpr,integerexpr

Description: The ASOUND command specifies the parameters for the voice, but does not actually start the sound. SCONTROL starts and stops all sound channels, allowing several voices to be synchronized. The first integerexpression controls voice 0, the second integerexpression controls voice 1, the third controls 2, and the fourth controls 3. If the integerexpression is 1, the voice is started or continued. If it is 0, the voice is stopped or not started.

SETARRAY
Type: BASIC command

Format: SETARRAY variablename,expression

Description: Sets each element of the array equal to the value of the expression. The variablename must be the name of a real or integer one dimensional array. If real, the expression must be real; if integer, the expression must be integer.

Example:

SETARRAY A%,0% sets all elements of A% to 0%.

SETCOLOR
Type: BASIC command

Format: SETCOLOR integerexpress,integerexpress,integerexpress

Description: The first integerexpression specifies the color register (must be 0, 1, 2, 3, or 4). The second integerexpression sets the color and the third integerexpression sets the brightness. Chapter 13 describes the relationship between these numbers and the colors produced. Note that the particular color register used depends upon the graphics mode. See Chapter 13 for a more complete description.

SETINT@
Type: BASIC command

Format: SETINT@ integerexpr,integerexpr,integerexpr,integerexpr

Description: Works with ATARI's display list interrupt capability. The first integerexpression is an identifying number (0 to 7), which refers to a given interrupt. The second integerexpression gives the display list linenumber where the interrupt occurs. Note that the first three display list lines (0, 1, and 2) are blanks, so that line 3 is the top display line. Also, the changes produced by the interrupt take effect at the start of the next screen line; that is, changes are not made in the middle of a line. The third integerexpression gives the memory location to be changed, and the fourth integerexpression gives the new value. To remove an interrupt, give the SETINT@ command with the last three integerexpressions equal to 0%. Before using SETINT@, you must append DLISTINT.APP. See CINT@ and Chapter 15.

SIN
Type: real function

Format: SIN(realexpression)

Description: Returns the sine of the value of the realexpression. Radians are assumed, unless the DEG command was executed.

SGN
Type: real function

Format: SGN(realexpression)

Description: Returns 1 if the realexpression is greater than 0, returns 0 if it equals zero, and -1 if less than 0.

SOUND
Type: BASIC command

Format: SOUND integerexpr,integerexpr,integerexpr,integerexpr

Description: Sets the sound for one of the four ATARI sound channels. The sound continues until an END command or another SOUND is executed for that channel. The first integerexpression sets the channel and must be 0, 1, 2, or 3. The second integerexpression determines the frequency (see table below). The third integerexpression (0 to 14) must be even and controls

distortion. 10 gives a pure tone. The fourth integerexpression sets the volume and must be less than 16 and greater than or equal to 0.

Integerexpression	Frequency	Integerexpression	Frequency
29	high C	91	F
31	B	96	E
33	A#	102	D#
35	A	108	D
37	G#	114	C#
40	G	121	middle C
42	F#	128	B
45	F	136	A#
47	E	144	A
50	D#	153	G#
53	D	162	G
57	C#	173	F#
60	C	182	F
64	B	193	E
68	A#	204	D#
72	A	217	D
76	G#	230	C#
81	G	243	C
85	F#		

SQR
Type: real function

Format: SQR(realexpression)

Description: Returns the square root of the value of the realexpression.

STICK
Type: integer function

Format: STICK(integerexpression)

Description: Returns an integer whose value depends upon the joystick position according to the following chart:

Function value	Joystick position
15	center
7	right
6	forward right
14	forward
10	forward left
11	left
9	back left
13	back
5	back right

Hint: If you take STICK(T%) and 3%, you can tell whether the stick is forward or back: 3 means neither, 1 means back, and 2 means forward. If you take STICK(T%) and 12%, you can tell whether the stick is left or

right: 12 means neither, 8 means left, and 4 means right. The value of the integerexpression determines the joystick, and must be 0, 1, 2, or 3.

Example: PRINT STICK(0%)

If joystick 0 is shifted right, the line prints 7.

STRIG
Type: BASIC function

Format: STRIG(integerexpression)

Description: Checks the fire button of the joystick given by the value of the integerexpression. If the button is pressed, STRIG returns zero; otherwise it returns 1.

STRING
Type: string function

Format: STRING(integerexpression,integerexpression)

Description: Creates a string whose bytes equal the value of the second integerexpression. The length of the string equals the value of the first integerexpression.

Example:

```
100 PRINT STRING(5%,ASC("A"))
RUN
AAAAA
```

STR$
Type: real function

Format: STR$(realexpression)

Description: Converts the number given by the realexpression into a string. For example, if the realexpression equals 1.2, STR$ returns a string whose first byte is 49 (ASCII code for 1), second byte is 46 (ASCII code for decimal point), and third byte is 50 (ASCII code for 2).

SUB SUBEND
Type: BASIC command

Format: SUB subroutinename
SUB subroutinename(variablename,...,variablename)

Description: Used to define a named subroutine. The subroutine may extend over more than one line and must end in a SUBEND command. Note that the SUB command must be the first statement on a line. Named subroutines make programs easier to understand; that is, FIGURETAX@ is more meaningful than GOSUB 2000. The subroutine is called when its name is used.

The subroutinename following SUB must end in @, telling the compiler that it stands for a named subroutine. The variables in parentheses are called dummy variables and no more than four are possible. They can be integer,real, or string. See Chapter 8 for more information.

TAB
Type: BASIC command

Format: TAB(integerexpression)

Description: Used with PRINT and LPRINT to tab to the print position specified by the integerexpression. If you follow a tab with a comma, what is printed next will not be at the tab position, but at the start of the next print zone. You must use a semicolon after a tab if you want the next printed item to start at the tab position. If you try to tab to a position to the left of the current print position, the tab will be ignored. Note that the leftmost print position is zero, and the one just to the right of it is 1. In the example, the left screen margin is set at 2, so the 'R' in RUN is at tab position 2.

Example:

```
100 PRINT TAB(3%);"A";TAB(6%);"B"
RUN
 A  B
```

TAN
Type: real function

Format: TAN(realexpression)

Description: Returns the tangent of the value of the realexpression. Radians are assumed, unless the DEG command was given.

Example:

```
100 A=1
110 PRINT TAN(A)
```

TIME
Type: integer function

Format: TIME

Description: Returns the value of the timer, which counts by one for each 1/60 second. Thus, if TIME returns 120, then 120/60 seconds have passed since the last reset of the timer. The longest time which can be measured is 32767/60=546.1 seconds, or about 9 minutes. See RTIME and WAIT.

TRAP
Type: BASIC command

Format: TRAP linenumber

Description: Tells the system what to do in case of an error. Once TRAP has been executed, any subsequent error causes the program to go to the line specified in the linenumber instead of returning to BASIC. Memory location 1240 will have the error number. After setting a TRAP, if you want to switch back and have errors cause a return to BASIC, give a TRAP 0 command. Appendix C lists the error numbers.

Example:

```
100 TRAP 200
110 OPEN "I",1%,"DATA"
120 TRAP 0
 .
 .
 .
200 PRINT "message"
210 INPUT ""T$
220 GOTO 110
```

If an error occurs in the opening of a file, the message on line 200 will be printed. The message specifies corrective action and tells the user to press RETURN. Then the program returns to line 110 and again tries to OPEN the file. Failure will cause another return to line 200. Success will reset the TRAP so that subsequent errors will cause a return to BASIC. You can set and reset the TRAP as often as desired.

VAL
Type: real function

Format: VAL(stringexpression)

Description: If the characters in the stringexpression represent a number, VAL returns this number. See STR$ and VAL%.

VAL%
Type: integer function

Format: VAL%(stringexpression)

Description: If the characters in the stringexpression represent an integer, VAL% returns this integer. See NUM$ and VAL.

WAIT
Type: BASIC command

Format: WAIT(integerexpression)

Description: Causes the program to pause for a number of seconds equal to the value of the integerexpression divided by 60. For WAIT to work properly, the timer sould not overflow; that is, a reset timer should have been given within the last 9 minutes. See RTIME and TIME.

WHILE WEND
Type: BASIC command

Format: WHILE condition
.
.
.
WEND

Description: Provides a way of looping through a sequence of statements. The program tests the condition. If it is false, execution continues at the statement immediately following WEND. If true, it executes the statements between WHILE and WEND. WEND returns control to the top of the WHILE loop, where the condition is again tested. FOR/NEXT is a more useful looping tool when you know how many times the loop must be executed. If you do not know this, WHILE/WEND and REPEAT/UNTIL are more useful.

Example:

```
100 READ A
110 WHILE A>=0
120     S=S+A
130     READ A
140 WEND
150 PRINT S
160 DATA 1,2,3,-1
RUN
6
```

Appendix A ASCII Code for ATARI

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
32	20	space	64	40	@	96	60	
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	
60	3C	<	92	5C	\	124	7C	
61	3D	=	93	5D	]	125	7D	clear screen
62	3E	>	94	5E	^	126	7E	backspace
63	3F	?	95	5F	_	127	7F	tab

Appendix B Reserved Words

ABS	HPOS	PUSING%
ABS%	IF	PUSING1%
ADR	INPUT	PUSING2%
ASC	INPUTLINE	PUSING3%
ASCB	INSERTB	PUT
ASCW	INSERTW	RAD
ASOUND	INSTR	READ
ATAN	INTSTR1	REM
CASE	INT	REPEAT
CHR$	LEFT	RESTORE
CHRW$	LEN	RETURN
CINT@	LET	RIGHT
CLOSE	LOADST	RND
CODE	LOCATE	RND%
CODEL	LOG	RTIME
COLL	LPRINT	SCONTROL
COLOR	LSCREEN	SETARRAY
COS	MACHINE	SETCOLOR
DATA	MID	SETINT@
DEF	NEXT	SIN
DEG	NUM$	SGN
DIM	OFFDISPLAY	SOUND
DFILL	ON	SQR
DO	ONDISPLAY	STEP
DRAWTO	OPEN	STICK
ELSE	PCONTROL	STRIG
END	PDISPLAY	STRING
ENDIF	PEEK	STR$
EOF	PEEKW	SUB
EXG	PLOT	SUBEND
EXP	POINT	TAB
FAST	POKE	TAN
FILL	POKEW	TIME
FINT	POPST	TRAP
FIX	POS	UNTIL
FOR	PRATE	USING
GET	PRINT	VAL
GETKEY	PSETCOLOR	VAL%
GOSUB	PSIZE	WAIT
GOTO	PUSING$	WEND
GRAPHICS	PUSING1$	WHILE

Appendix C Error Messages

1 or 2 SYNTAX ERR Several errors can cause this message.

3 MISSING LINE A reference was made to a non-existent line.

4 MISSING QUOTE Quotes missing from a string expression.

5 ARGUMENT ERR Several errors can cause this message. You might have forgotten to dimension an array or used the DIM statement after the first use of an array. The arguments in a function or named subroutine may differ from the way the function or subroutine was defined. Remember, in user-defined functions and subroutines, the system will not convert between integers and real numbers.

6 UNDEF.FN,SUB,OR ARRAY You forgot to define a function or subroutine or to dimension an array. Another possibility is a spelling mistake.

7 REDEF.FN,SUB,OR ARRAY You defined a function or subroutine twice, dimensioned an array twice, or used an array before the DIM statement.

11 MUST BE AT LINE START DATA,SUB, and DEF statements must be first in a line; the statement 100 A%=Z%: DATA 5 gives this error.

12 MAX.NUMB.EXC. Maximum number of variables (255) or maximum number of lines (975) exceeded. If maximum number of variables exceeded, I suggest you try executing CLEAN.COD to remove unused variables, if any.

13 ARITHMETIC ERR. Possibilities include you divided by zero, or tried to create a floating point number larger than or equal to 10^99 or smaller or equal in absolute value to 10^-99.

14 EXP.TOO COMPLEX There is a maximum level of complexity possible in arithmetic expressions and a limit to the depth of nesting in loops. An error in one or both of these areas could cause this message.

15 STACK ERR. Possibilities include using POPST without LOADST or vice versa, thus exceeding the stack limit.

16 OUT OF DATA You have tried to read more data than is present in the program.

17 MEMORY EXCEEDED See Chapter 11.

18 NUMBER OUT OF RANGE The most likely cause is that you exceeded the maximum possible subscript value in an array. It is also possible that you used a negative subscript, or that in a string function, you used a number beyond the length of the string.

19 NO GOSUB Your program tried to execute a RETURN, but had never executed a GOSUB.

20 NO FOR You used a NEXT with no matching FOR. Note that this could be a spelling error. Unlike some BASICs, Advan checks the variable in the FOR and NEXT to make sure they agree.

21 NO CASE You used CASE END,CASE ELSE, or & with no CASE. All CASE

statements must start with a CASE command.

22 NO IF The program encountered an ENDIF or ELSE in what appeared to it as a multi-line IF/DO, but had no matching IF/DO.

23 NO WHILE You used WEND with no matching WHILE.

24 NO REPEAT You used UNTIL with no matching REPEAT.

25 NO SUB You used SUBEND with no matching SUB. Remember, only one SUBEND can be used in a named subroutine.

26 NO NEXT You used FOR with no matching NEXT. Remember, Advan BASIC checks the variablename in FOR/NEXT. If they do not match, an error message is given.

27 NO CASE END You used CASE with no matching CASE END.

28 NO ENDIF You used IF/DO with no matching ENDIF.

29 NO WEND You used WHILE with no matching WEND.

30 NO UNTIL You used REPEAT with no matching UNTIL.

31 NO SUBEND You used SUB with no matching SUBEND.

32 FILE OPEN You may have tried to open a file which was already open, or you may have tried to open two files with the same file number.

33 NO SUCH DEV. You tried to open a non-existent disk or a non-disk device. In Advan BASIC, OPEN refers only to disks.

34 WRITE ONLY You tried to read from a file opened for input only.

35 NOT OPEN You tried to GET, PUT, etc. to a file which is not open.

36 READ ONLY You tried to write to a file opened for READ ONLY, or you tried to OPEN in output or append mode a locked file or a file on a write protected disk.

37 END FILE You tried to read past the end of a file.

38 DISK FULL There is no more room on the disk.

39 FILE LOCKED You tried to erase a locked file. A file must be unlocked before it can be erased.

40 DIR FULL Only 64 files are possible on a disk. You tried to exceed that number.

41 FILE NOT FOUND The system could not find the file on the specified disk.

42 I/O ERROR Normally occurs when something is wrong with the disk or disk drive.

Appendix D Memory Map of the BASIC

0000 to 00D3 reserved for Advan BASIC as working storage

00D4 to 00FF used by Advan BASIC, but can be used by machine code

0100 to 06FF reserved for Advan BASIC

0700 to 1DAF Advan DOS

A000 to BFFF Advan execute module

C000 to FFFF ATARI operating system

If graphics mode 0 is used, the display occupies the region from 9C1C to 9FFF. BASIC and the user program occupy the region from 1DB0 to 9C1B. Note that in an XL or XE machine, 14K of the BASIC is transferred to upper memory during program execution.

If a graphics mode other than 0 is selected, 8000 to 9FFF is set aside as a display region, 7C00 to 7FFF for players, 7800 to 7AFF for an alternate character set, and 7B00 to 7BFF for missiles or for the top part of an alternate character set. The region from 1DB0 to 77FF is used for the program and the part of the BASIC not transferred to upper RAM.

Special Locations

Hex	Decimal	Description
54	84	row number of current cursor location
55,56	85,86	column number of current cursor location (use PEEKW(85%) to get column number)
52	82	contains column number of left screen margin
53	83	contains column number of right screen margin
2F0	752	cursor display control flag. POKE 752%,1% removes cursor; POKE 752%,0% restores cursor. Note that the cursor is not changed until the next print command.
290	656	row number of split screen text cursor
291,292	657,658	column number of split screen text cursor
4E3	1251	printer width

Appendix E Assembly Language Mnemonics used by Advan BASIC Compiler

In order to understand the mnemonics used by Advan BASIC, consider the ADC command. This command causes a number and the carry bit to be added to the accumulator. The question is, where is the number? Advan BASIC uses extensions to the command to specify the location of the number.

ADC The two bytes following ADC specify the location of the number. For example, ADC,FF,9F adds the number at 9FFF to the accumulator.

ADCZ The single byte following ADCZ specifies the zero page location of the number. For example, ADCZ,E0 adds the number in 00E0 to the accumulator.

ADCIM The single byte following ADCIM will itself be added to the accumulator. For example, ADCIM,2 adds 2 to the accumulator.

ADCX The X register is added to the two bytes following the ADCX command. This gives the location of the number to be added to the accumulator. For example, ADCX,3,2 (with x=2) adds the number stored in 205 (i.e., 203+2) to the accumulator.

ADCY Same as ADCX, except the Y register is used instead of the X register.

ADCIY The single byte following ADCIY specifies the zero page location of the two byte address. The Y register is added to the two byte number in page zero to get the address of the number to be added to the accumulator. For example, consider ADCIY,E0. The two byte address is at 00E0 and 00E1. The number stored in 00E0 and 00E1 is added to the Y register to form the address.

ADCIX The single byte following ADCIX is added to the X register. The sum (must be in zero page) is the location of the first byte of the two byte address. For example, consider ADCIX,E0. If the X register=4, the address of the number to be added to the accumulator is 00E4 and 00E5.

ADCZX The single byte following ADCZX is added to the X register. This is the address (in zero page) of the number to be added to the accumulator. For example, consider ADCZX,E0. If the X register=4, the number to be added to the accumulator is in memory location 00E4.

In addition, the following two commands show the last two possible extensions.

ASLA Takes the number in the accumulator and shifts it left.

LDXZY The single byte following LDXZY is added to the Y register. The sum is the zero page location of a number which is loaded to the X register.

The following table lists the possible 6502 commands by mnemonic.

ADC, ADCIM, ADCIX, ADCIY, ADCX, ADCY, ADCZ, ADCZX get the number from the specified location and add with carry to the accumulator

AND, ANDIM, ANDIX, ANDIY, ANDX, ANDY, ANDZ, ANDZX get the number from the

specified location and then AND it with the accumulator

ASL, ASLA, ASLX, ASLZ, ASLZX shift specified number left by one bit

BCC branch on carry clear

BCS branch on carry set

BEQ branch if result is zero

BIT, BITZ compare bits in accumulator with those in the specified memory location

BMI branch if result is negative

BNE branch if result is not zero

BPL branch if result is positive

BRK forces break

BVC branch if overflow not set

BVS branch if overflow set

CLC clear carry flag

CLD clear decimal mode

CLI clear interrupt flag (enables interrupts)

CLV clear overflow flag

CMP, CMPIM, CMPIX, CMPIY, CMPX, CMPY, CMPZ, CMPZX compare number at specified memory location with accumulator

CPX, CPXIM, CPXZ compare mumber at specified memory location with X register

CPY, CPYIM, CPYZ compare number at specified memory location with Y register

DEC, DECX, DECZ, DECZX decrement by one the number in the specified memory location

DEX decrement X index register by one

DEY decrement Y index register by one

EOR, EORIM, EORIX, EORIY, EORX, EORY, EORZ, EORZX exclusive OR the accumulator with the number in the specified memory location

INC, INCX, INCZ, INCZX increment by one the number in the specified memory location

INX increment X index register by one

INY increment Y index register by one

JMP JUMP to new address

JSR go to a subroutine

LDA, LDAIM, LDAIX, LDAIY, LDAX, LDAY, LDAZ, LDAZX load number from specified memory location to accumulator

LDX, LDXIM, LDXY, LDXZ, LDXZY load number from specified memory location to the X register

LDY, LDYIM, LDYX, LDYZ, LDYZX load number from specified memory location to the Y register

LSR, LSRA, LSRX, LSRZ, LSRZX shift the number in the specified location to the right by one

NOP no operation

ORA, ORAIM, ORAIX, ORAIY, ORAX, ORAY, ORAZ, ORAZX OR the accumulator with the number in the specified memory location

PHA push accumulator to the stack

PLA pull number from stack and put into the accumulator

PLP pull number from stack and put in processor status register

ROL, ROLA, ROLX, ROLZ, ROLZX rotate number in specified location left by one

ROR, RORA, RORX, RORZ, RORZX rotate number in specified location right by one

RTI return from interrupt subroutine

RTS return from subroutine

SBC, SBCIM, SBCIX, SBCIY, SBCX, SBCY, SBCZ, SBCZX subtract number at specified memory location and borrow from accumulator

SEC set carry bit

SED set decimal mode

SEI set interrupt flag (disable interrupt)

STA, STAIX, STAIY, STAX, STAY, STAZ, STAZX store accumulator at the specified location

STX, STXZ, STXZY store X register at specified memory location

STY, STYZ, STYZX store Y register at specified memory location

TAX transfer accumulator to X register

TAY transfer accumulator to Y register

TSX transfer stack pointer to X register

TXA transfer X register to accumulator

TXS transfer X register to stack pointer

TYA transfer Y register to accumulator

Appendix F Reporting Problems or Errors

If you should encounter a problem or error in the BASIC or the manual, we would appreciate hearing about it. Please list the computer you were using and, if possible, a short example of a program which malfunctions. Send to:

Advan Language Designs
P.O. Box 159
Baldwin, Kansas 66006

INDEX

Addendum for Advan BASIC Compiler Manual

Page 30 Add:

While performing string operations the system uses part of the memory as work area. After a certain number of string operations, this area will need to be reorganized. To do this, the system stops the execution of your program briefly (usually a second or less), cleans the area, and resumes program execution. In many cases, you can, if necessary, virtually eliminate this pause. The command CODE"8A" inserted into program will force the system to reorganize the area. If this command is executed at regular intervals, the system will not have to pause to clean the area. The length of time to reorganize depends upon the number of string operations performed since last reorganization. Thus if only a few string operations take place between the execution of CODE"8A" commands, the time required for each of them will be very short. However, overall execution speed will be reduced somewhat by this technique, since it is more efficient to reorganize only when necessary.

Page 31 At end of SAVEC and EXEC section, add:

Normally if an error occurs during execution of a program, the system will give the error message and then display the program line where the error occurred. Suppose, however, you are using the EXEC command to run a compiled program from the disk. In this case, the program is not in the computer and the message 'INSERT PROG. DISK&ENTER NAME' will be given. At this point you should insert a disk with the program and enter the program name. The system will find the line on which error occurred and will print out the linenumber as well as the error message.

Page 49 At the end of the page, add:

Special note: Players and missiles are moved and/or changed during the vertical blank interrupt. Horizontal movement requires little computer time; vertical movement and/or changing the figure requires more time, especially for missiles. The greater the figure's height the longer the time. Since there is a limited amount of time available at the interrupt, it is important to keep the figures as short as possible. If too much time is taken, figures at the top of the screen might be distorted and some might not be moved.

Page 58 At the end of the SIEVE.BAS section, add:

Since this program needs a large amount of memory, 400/800 users will need to type RUN SIEVE.BAS 2.

New System Commands

Since the first release of Advan BASIC two new system commands have been added. The description of these commands should be added to the list on page 64 of the Advan BASIC manual.

INTEGER
Format: INTEGER

Description: Variable names normally represent real (floating point) numbers unless the name ends in % or $. Also numbers will be in real form unless the number ends in %. In some programs you want most of the variables and numbers to be in integer form, and the repeated typing of % becomes a chore. If you use the INTEGER command before entering a program, however, the naming convention for numbers and variables is changed. Variable names with represent integers unless they end in ! or $. Names representing real numbers must end with !. - In addition, numbers will represent integers unless they have a decimal point or an E in them. For example, consider the following program:

```
10 T%=1%:S!=1.:FOR Y=1 TO 2000:S!=S!+T%+Y:NEXT Y
```

If you use the INTEGER command before entering the above program, the Y, 1, 2000, T%, and 1% will be integers and the S! and 1. will be real. If you do not use the INTEGER command, T% and 1% will be integers and Y, 1, 2000, S!, and 1, will be real.

Special Notes:
(1) When you give the INTEGER command, the system will change the border color so that you always know when you are in the INTEGER mode.
(2) If a program is already in the computer, the system vi11 not accept the INTEGER command. If you forget to give the INTEGER command before entering a program , you can use the SAVES command followed by the NEW command and then the INTEGER command. Finally, reload the program with the LOADS command.
(3) If you use the SAVE command the information about whether you are in the INTEGER mode will be saved with the program. When you use the LOAD command the system will automatically shift to the correct mode.

REAL
Format: REAL

Description: Returns the system from INTEGER to normal node. The border will also be returned to black, Like the INTEGER command, REAL cannot be used if there is a program in the computer. If you enter a program using INTEGER mode and you want to change it to normal (real) mode, use the SAVES command followed by the NEW command and then the REAL command. Finally reload the program with the LOADS command.

CONVERT PROGRAM

Advan BASIC is a fast, powerful, efficient BASIC which takes advantage of ATARI'S excellent graphics and sound capabilities. Thus, if you are planning fo write a new BASIC program for the ATARI, you will normally get a faster and better program using Advan BASIC rather than ATARI BASIC. However, any Advan owners might want to use Advan to speed up existing ATARI BASIC programs which they have written. Since Advan BASIC cannot directly load and run these programs, they either need to be rewritten or converted. If you want to take fu1l advantage of Advan's speed and power, the best approach is to rewrite these programs following the same general program outline but using the full range of Advan commands. If you want to try to speed up your program without rewriting, however, we have added to our package a convert program which will convert many (but by no means all) ATARI programs to Advan format. Note that it is not always an advantage to speed up a program. For example a game might become too fast to play, or sounds may no longer sound right.

To use the convert program you must be in Advan BASIC. Insert a disk with CONVERT.COD into drive 1. This program along with CONVERT.SUB is on the Master disk. You can use a Master disk or you can use COPYFILE.COD to transfer one or both programs to another disk and use that disk (see Ch. 17 of. the Advan BASIC manual). Now type EXEC CONVERT.COD. First, you are asked for the name of the ATARI program. This program should have been saved to a disk using the SAVE command while in ATARI BASIC. Insert the disk with the program and then enter the name. Then you are asked for the name of the Advan program to be created; enter that name.

Next you are given the opportunity to change one or more of the variables used in your program from real to integer. (For advantages of integers see Ch.2 of manual.) A list of the variables is displayed. If the variable represents a real array, it will have a period inserted into the name. To change all real variables to integers, type % and then press RETURN. To change one variable, type the variable name and press RETURN. The variable will be changed and the list displayed again, allowing you to make another change. When you have switched all the variables you desire to integer form, press RETURN and the conversion will begin. Ready is displayed when the conversion is completed.

Now insert a disk with CONVERT.SUB on it and type LOAD CONVERT.SUB. This loads the subroutines required to handle the differences in the BASICs. Next reinsert the disk with the Advan form of the program and type APPEND followed by the name you gave to the Advan program. When the APPEND is completed type COMPILE 1. At the cornpletion of the compile type SAVEC followed by the name you used for the Advan program. This will place the compiled code on the disk. You can now execute the program by typing EXEC followed by the Advan program name.

As an example assume that the following ATARI BASIC program has been saved using the name ALPHA: 10 FOR T=1 T0 1000:S=S+T:NEXT T:? S:END

The following sequence will produce a program called BETA which Advan BASIC can run. Note that you ust be in Advan BASIC and that what you type

is underlined and what the computer displays is not. Comments are in parentheses.

```
EXEC CONVERT.COD (Disk with CONVERT.COD must be in drive 1)
Enter ATARI program name ALPHA (Insert disk with program before pressint
RETURN)
Enter Advan program name BETA

T S (A11 the variables used in the program are listed)

Enter name to change var. to int.
Enter % to change all real to int.
Press RETURN when done (In this program we do not want to convert either
variable to integer so press RETURN)
Ready (Advan program named BETA is now on disk)
LOAD CONVERT.SUB (Disk with CONVERT.SUB must be in drive 1)
Ready
APPEND BETA (Insert disk with BETA before pressing RETURN)
Ready
COMPILE 1
SAVEC BETA
EXEC BETA (This executes the converted program)
```

Some programs are so long that the above sequence won't work and you will get a NO ROOM error. For many of these programs the following sequence will work. Proceed as above through the LOAD CONVERT.SUB command and then:

```
SAVE CONVERT.SUB (Insert disk with BETA before pressing RETURN)
Ready
LOAD CONVERT.SUB 1
Ready
APPEND BETA
Ready
SAVE BETA
Ready
COMPILE BETA/BETA.COD
Ready
KILL BETA
EXEC BETA.COD (This executes the converted program)
```

The ATARI commands USR, FRE, PADDLE, STATUS, NOT, PTRIG, and LIST cannot be converted and so any program using them will need to be modified before conversion. In ATART BASIC the POP command can be used to remove information on the last GOSUB or to remove FOR loop data. The former use of POP is supported. The latter is neither supported nor needed. When a POP is encountered by the convert program, the line number will be listed so that you can check it. If it is used for FOR loop data you will need to remove the POP command from the ATARI BASIC program. Of the XIO commands only the special fill command (i.e., #18) is supported.

In ATARI BASIC you can use a variable to set the dimension of an array or the length of a string. When the DIM statement is executed the value of the variable determines the dimension used. In Advan BASIC this is not

possible. The convert program will attempt to determine the value of the variable. If it cannot, the dimension will be set to 100 which is usually large enough. In either case the program will list the line number, the name of the variable being dimensioned, and the value that the dimension has been set to. If the value is not appropriate you will need to change the ATARI program.

ATARI BASIC also permits variables to be used for the line numbers in GOTO and GOSUB statements. This would be difficult for any compiler to support and Advan does not. Again the convert program will attempt to determine the value of the variable. It will list the line number and the value used or give an error if it cannot determine a value. Also some of the programming tricks used in ATARI BASIC will give an error in Advan BASIC. For example, if you put NEXT statements in IF commands you can use several NEXT statements with each FOR. This will give an error in Advan BASIC and you will need to modify the program before it can be converted. Note that if you are using player missiles you might have to add 64 to the number used in the GRAPHICS commands.